An Algebraic Model of Software Evolution

by

Benjamin J. Keller

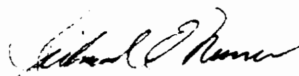
Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

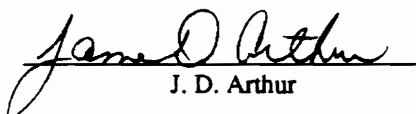
in

Computer Science

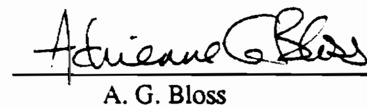
APPROVED:



R. E. Nance, Chairman



J. D. Arthur



A. G. Bloss

D. C. S. Allison

September, 1990

Blacksburg, Virginia

LD
5655
V855
1990
K454
c.2

An Algebraic Model of Software Evolution

by

Benjamin J. Keller

Committee Chairman: Richard E. Nance

Computer Science

(ABSTRACT)

A model of the software evolution process, called the Abstraction Refinement Model, is described which builds on the algebraic influence of the Laws of Programming and the transformational Draco Paradigm. The result is an algebraic structure consisting of the states of the software product (system descriptions) ordered by a relation of relative correctness with transformations defined between the system descriptions. This structure is interpreted as the software evolution space, with the intended semantics given by a model combining axiomatic semantics and the Lindenbaum algebra of a first-order logic. Using this interpretation, software evolution can be represented as a sequence of transformations on system descriptions.

The primary contributions of the characterization of software evolution are to the understanding of maintenance and its relationship to development. The influence of development on maintenance is shown as the transfer of a "descriptive context" for the software system. This context is used as an information source during maintenance, and is progressively modified by maintenance activities. These activities are characterized by balanced forward and reverse transformations. The use of reverse transformations explaining the role of reverse engineering in maintenance for information gathering and document reconstruction. Additionally, the form of maintenance affects the performance of the activity, with adaptive maintenance differing from corrective, perfective and preventive maintenance. These factors contribute to the descriptive nature and utility of the Abstraction Refinement Model in defining methodologies for maintenance.

Acknowledgements

I come out of my Master's studies with significantly different interests than when I started. The two people who get most of the blame are Dr. John Roach and Dr. Dick Nance. Dr. Roach helped kindle my interest in algebra and logic, and taught me the merit of filling my mathematical "tool-belt." Dr. Nance introduced me to the problems of software engineering[1], and more importantly has guided the development of the ideas presented here. Without this combination my philosophy would undoubtedly be more orthodox, but slightly less interesting.

I am also indebted to those people who have supported my exploration of computer science. This includes Dr. James "Sean" Arthur who has made a significant impact on this research through his supposedly "stupid questions;" Dr. Adrienne Bloss who helped me understand programming language semantics; and Dr. Lenny Heath who tolerated my blunders through complexity theory and has provided the occasional prod and moral support. Additionally, this includes Dr. Robert Crawford and Dr. Thomas Cheatham of Western Kentucky University who helped me get into this mess in the first place.

[1]Although not a direct result, much of the research presented in this thesis was motivated by my involvement in projects directed by Dr. Nance for the Naval Surface Warfare Center. I am grateful for both the monetary support and inspiration that this work has provided.

Contents

Chapter 1. Maintenance and Models of Software Evolution 1

Chapter 2. Literature Context 4

2.1. Program 4

2.1.1. Program versus Specification 5

2.1.2. Abstraction and Non-determinism 6

2.1.3. Wide- versus Narrow-Spectrum Languages 9

2.2. Programming 9

2.2.1. Models of Programming 9

2.2.2. Methodologies 20

2.2.3. Environments and Tools 25

2.3. Program Execution 26

2.4. Summary 26

Chapter 3. Foundations 28

3.1. Mathematical Preliminaries 28

3.1.1. Lattice Theory 28

3.1.2. Lindenbaum Algebra 31

3.2. Semantic Foundations 32

3.2.1. Predicate Transformers 32

3.2.2. Sample language 36

3.2.3. Orders 39

3.3. Syntactic Foundations 40

3.3.1. Abstraction Ordering 40

3.3.2. Join and Meet 41

3.3.3. Programing Operators 42

3.4. Foundational Synthesis 43

3.4.1. Observations 43

3.4.2. Assumptions 46

Chapter 4. Model Development . . . 48

4.1. Intended Semantics . . . 49

4.1.1. Informal Discussion . . . 49

4.1.2. Formal Definition . . . 51

4.2. Syntactic Development . . . 59

4.2.1. Construction of Semantic Structure . . . 59

4.2.2. Transformations . . . 62

4.3. Summary . . . 67

Chapter 5. Software Evolution Process Characterization . . . 68

5.1. Development . . . 70

5.1.1. Analysis Phase . . . 70

5.1.2. Transformational Phase . . . 72

5.2. Maintenance . . . 77

5.2.1. Analysis Phase . . . 77

5.2.2. Transformational Phase . . . 78

5.3. Complete Life-Cycle . . . 88

Chapter 6. Conclusions . . . 91

6.1. Evaluation of Model Contributions . . . 91

6.1.1. Relate Maintenance to Development . . . 91

6.1.2. Non-Methodological . . . 92

6.1.3. Recognize Imperfections . . . 92

6.2. Extensions for Model Improvement . . . 93

6.2.1. Hoare's Notion of Specification Language . . . 93

6.2.2. Narrow-spectrum Issues . . . 94

Bibliography . . . 96

Chapter 1. Maintenance and Models of Software Evolution

While practitioners are faced with the problems of keeping aging software systems functional and up-to-date, research in the field of software engineering has produced limited and disconnected knowledge of the software maintenance process. The ability to handle the maintenance of software systems efficiently and effectively is crucial to improvement in their quality over extended lifetimes. Despite this fact, a survey of the literature indicates that research has centered on the development activity to such a degree that software maintenance is described as "a relatively neglected subject" by Swanson and Beath [SWAE90].

Representative of this emphasis on development are the so called life-cycle models. Most consider maintenance as an afterthought, either by lumping maintenance activities into a single activity or not mentioning maintenance at all. Models of the second type either impose an assumption that maintenance activities are identical with development and performed in the same order as in development, or ignore the need for precise definition and structuring of maintenance activities. Typically a model of this sort imposes a methodology on the software evolution process.

The term *software evolution process* recognizes the import of the ideas of Lehman and Belady [LEHM85]. Software systems are considered to change continuously during their life, and by this change evolve to essentially different systems. The *process* of software evolution is the collection of activities or actions taken which cause the evolution [DOWM86]. This is a more general notion than life-cycle, since a life-cycle model implicitly assumes that certain activities recur in a specific order throughout the life of the system. A model of the software evolution process might reveal more about the nature of the process, because it represents what happens and not what *should* happen.

The purpose of this research is to define such a model of software evolution. This model should avoid the deficiencies associated with life-cycle models with respect to maintenance. In addition, the model should help to differentiate appropriate and inappropriate activities in the evolution of a system. The goals of the model are explicitly stated below:

(1) Relate Maintenance to Development

The model should present the software evolution as a "continuous" process. Further, the model should be able to describe what is known of both maintenance and development. In addition, the model must be able to show how the two activities are related. In this way

development effects on maintenance, and the effects maintenance has on future maintenance can be better understood.

(2) Non-Methodological

In answer to the second criticism of life-cycle models, the model developed must not be constrained by methodology. In other words, the model should not mandate a particular methodology. Any conceivable methodology, effective or not, should be characterizable in the formalism of the model.

(3) Recognize Imperfections

The model should allow all possible scenarios of evolution. Both good and bad activities and products should be represented to underscore the requisite maintenance needs and methodology requirements.

To meet these goals, the governing philosophy is that a software evolution model should portray all possible states of a software product as well as the state transitions. The term *software product* refers to the final realization of the software system – the implementation in a programming language – and the corresponding documentation. A *state of the software product* is any of the intermediate realizations of the system (specification, design, implementation, etc.). Alternatively, states are called *product realizations* or *system descriptions.*

The portrayal of these product states and state transitions only gives a partial representation of the software evolution process. Missing is the ability to represent the quality of the product realizations and transitions. The quality of interest is correctness of one product with respect to another, and a "satisfies" relation can be used to express this quality. A complete model should portray:

(1) the state of the product,
(2) the "satisfies" relationship between product realizations, and
(3) the transitions between product realizations.

While the relationship of relative correctness is expressed by the "satisfies" relation, other aspects of quality (reusability, robustness, adaptability, etc.) are not directly impacted by this relation.

A model intended to meet these requirements, the Abstraction Refinement Model (ARM), is developed in this thesis. Such a model has a substantial context within the literature, and this is presented in Chapter 2. The discussion is organized to emphasize the concepts which most influence

the model. The third chapter emphasizes the foundational material, and the fourth contains the formal development of the model, presenting the syntactic and semantic aspects of the requirements. Chapter 5 provides an interpretation of the model in terms of software evolution. The capabilities of the model with respect to the criteria stated, and possible extensions are summarized in Chapter 6.

Chapter 2. Literature Context

A problem arises in trying to present the literature which influences a software evolution model, particularly for a model like the Abstraction Refinement Model. The difficulty lies in giving proper emphasis to those aspects of the literature which have the most relevance. The following discussion is focused on three main subjects: "program," "programming," and "program execution."

The choice of the terms "program" and "programming" might be criticized. Others might want to see these as "product" and "process" to represent greater generality, but these seem too abstract from the issues of interest. Within this chapter the term *program* refers to a representation of a computation, and the term *programming* refers to the construction or modification of such a representation.

The division of the literature by "program," "programming" and "program execution" to some extent represents the trichotomy of project, process, and product [ARTJ86]. Considering only development for a moment, a software development project is intended to build a software system with a certain behavior (represented by program execution). To achieve this system a construction process (programming) begins which results in the product (program). The division, therefore, is rooted in the description of software evolution.

2.1. Program.

One requirement imposed in the first chapter for a complete model of software evolution is that it be capable of portraying the product state. This means that all representations of the system should be possible within the formalism of the model. The essential aspect of this representation is the issue of abstraction.

The emphasis on abstraction in this section is motivated by two of the requirements stated in the first chapter. The first is mentioned above: representation of the state of the product. The second is the portrayal of the "satisfies" relationship between realizations of the system. These requirements imply the need for (a) abstract formalisms, and (b) formalization of the specific relationship of one program being more abstract than another.

Abstraction is a fundamental principle of software development [ARTJ86] and also of software maintenance. Abstraction is used in the preliminary stages of development during the statement of requirements for a software system. This specification must simultaneously hide the application

domain details from the programmer and the implementation issues from the "customer." The refinement of this abstraction leads to the realization of the specification as an executable system.

This section provides a perspective on the treatment of abstraction and related issues in the literature. First, different approaches to programs and specifications are presented. Second, the conceptual and formal relationships between abstraction and non-determinism are explored. Third, a division of programming languages by the degree to which they express abstraction is discussed.

2.1.1. Program versus Specification. The most concrete example of abstraction is the distinction between programs and specifications. The properties of programs have been stated by Dijkstra [DIJE76]. The extension to specification is made by dropping some of these properties, and is discussed in many places [MORJ89] [BACR89] [MORC88] [NELG89]. The relationships between specifications and programs are elaborated in Chapter 3.

Dijkstra's characterization of programs is given in terms of pre- and post-conditions and predicate transformers. The *pre-* and *post-conditions* are predicates on the program variables. The pre-condition indicates the values of the variables allowed as input, and the post-conditions those values allowed as output.

The program syntax used by Dijkstra [DIJE76] is that of a non-deterministic procedural language. A specification language can be formed by extending the program syntax by *prescriptions* (pre-, post-condition pairs), as is done by Back [BACR89], Morgan [MORC88], and Morris [MORJ89]. The addition of prescriptions forces the dropping of two of Dijkstra's properties of programs.

In contrast, the syntax of the specification language used in the Laws of Programming of Hoare is simply a non-deterministic language. These "Laws" specify an equational theory of non-deterministic programs which allows the syntactic manipulation of programs. An abstraction order is used to represent the semantic relationship held between a specification and its realizations (i.e. the "satisfies" relation), and the equivalence of two programs [HOAC87][HOAC89]. In Chapter 3 the Laws of Programming are discussed further and are used as a basis for developing the ARM.

Other possible syntax for formal specifications includes functions which map input states to correct output states and binary relations which relate the input and output states [MILA86]. These forms of syntax seem unnecessarily difficult to comprehend. It is only necessary that specifications consist of "abstract commands" as defined by Hoare. These commands denote the "general properties of the desired behavior" of the system, without including details of implementation [HOAC85].

Specifications need not be satisfiable by an executable system [HOAC87]. During creation of the specification, operations from the application domain might be included which have no direct implementation (i.e. no routine can be written which computes the operation). While this introduces the problem of approximating the operation during implementation, it removes the concern with computability issues during analysis. These can be seen in the specifications formed by relaxing Dijkstra's properties of programs (see Chapter 3).

2.1.2. Abstraction and Non-determinism. In the last section two forms of specification languages are discussed. One is an extension of program syntax by the addition of prescriptions as a form of abstraction. The other is a non-deterministic language, where non-determinism serves the role of abstraction. The correspondence between abstraction and non-determinism is explored in this section. This correspondence is important because, like the work of Hoare et al. [HOAC87], the development of the Abstraction Refinement Model uses non-determinism as an explicit form of abstraction.

2.1.2.1. Conceptual Connections. The connection between abstraction and non-determinism is somewhat of an inverse relationship. While abstraction is characterized by ambiguity or lack of detail, non-determinism can be characterized by a choice of abundant detail. Before discussing the connection further, both abstraction and non-determinism need to be defined.

Abstraction can be used in three ways. As a noun, "abstraction" refers to a description in which certain details of the system are suppressed (while others may be emphasized) [SHAM84]. In this sense we can also relate two system descriptions by saying one is an abstraction of another. As a verb, "abstraction" refers to the process of suppressing detail. It is the suppression of detail which induces ambiguity in the description. Instead of providing the exact details of a computation only a vague description is given.

Non-determinism is a more explicit and better understood concept than abstraction. The use of non-determinism arises primarily in the study of languages for concurrency, examples being CSP [HOAC78] and CCS [MILR80]. Zave [ZAVP89] identifies five forms of non-determinism: scheduling, action, function, interactive and stream.

Scheduling non-determinism occurs in languages for which concurrent processes are part of the semantics. The non-determinism comes from the lack of need to know when such processes execute relative to each other until synchronization is required. This type of non-determinism is not strictly relevant to the issues of system design except for correctness (avoidance of deadlock, etc.).

Action non-determinism is the most explicit form and is the one related to abstraction. Also called "choice non-determinism" [CLIW81], action non-determinism is given explicitly by "choice points" within a program. Each choice is resolved by an external agent which selects from the alternatives [CLIW81]. This form of non-determinism is closely related to guarded commands and the weakest pre-conditions [ZAVP89] (foreshadowing of the next section).

Function non-determinism results from a lack of definition of a function. The non-determinism is resolved by selecting for a value in the domain a corresponding value from the range. This is much weaker than modeling such a function by a choice of different functions with the same domain and range (action non-determinism), because more selections are possible. (However, an explicit choice of all functions with the domain and range would be equivalent.) Notice that this form of non-determinism may result in a function "behaving" like a relation since several values might be produced for a single input each time it is given.

Interactive and *stream non-determinism* are closely related. Each views the input process as a choice over the possible range of input values. The two are distinguished by synchronization details [ZAVP89].

All forms of non-determinism are important, but the one of primary concern is action non-determinism, which is typically referred to in the literature. In the sequel, action non-determinism is simply called non-determinism.

The connection between non-determinism and abstraction is indicated by the phrase "ambiguity as choice." The implication of this phrase is that the ambiguity of an abstract system description can be replaced by a choice of details. This is similar to the argument that a conjunctive specification is more abstract than its constituent parts.

Making the abstraction of a system description explicit through the use of non-determinism is certainly not realistic. Each possible realization would have to be found and then composed via the non-determinism operator. Despite the impracticality of such an endeavor, the result would appear to be equivalent to the original system description. So, the claim is made that non-determinism can be used to explicitly represent abstraction. This claim could be justified through the connections between semantics for abstraction (predicate transformer semantics) and non-determinism.

2.1.2.2. Formal Connections. Semantic models are needed to explore the formal connections between abstraction and non-determinism. While these models are abundant for non-determinism (two are presented), they are rare for abstraction. An assumption is made that the predicate transformer

semantics is suitable for modeling abstraction. This assumption is supported by the definition of the specification language discussed earlier in which the programming language is extended by the addition of pre- and post-condition pairs.

2.1.2.2.1. Semantics of Abstraction. The semantics of abstraction are defined as the semantics of the prescription-based specification language discussed earlier. Primary to this definition is the definition of an abstraction or implementation order which indicates that one program is an abstraction of another. The definition given in Chapter 3 is a combination of that of Morris [MORJ89] and Back [BACR89], although others have made similar definitions [MORC88] [NELG89]. (Nelson suggests a correspondence to power domains so the differences in approach are noted.)

2.1.2.2.2. Semantics of Non-determinism. The semantic models of non-determinism are varied. Perhaps the oldest and easiest to understand is the relational model [NELG89], but the power domain model of denotational semantics is more recent. Both approach the problem of representing the inexact behavior of non-deterministic programs by functions.

The relational model represents the multiple outcomes of a non-deterministic program by binary relations on states and outcomes. The set of outcomes may simply be the set of states or may also include an element $\perp$ which represents infinite looping (such a model is "lifted" [SCHD86]). The relations may be restricted to being total, so that each has an outcome associated with it [NELG89]. A lifted model might also allow non-termination as a possible initial state, but only allow it to relate to itself (strictness) [MAIM89].

In denotational semantics a deterministic program is represented by a function over some domain (a type of partially ordered set [CLIW81]). The power domain is an extension suggested by Plotkin [PLOG76] which represents outcomes as sets of values. So instead of mapping to a particular value, the function maps to a set of values.

Although the relational model has the closest intuitive relationship to axiomatic semantics, power domain models have been formally related. Other models have been introduced as well; two examples (not discussed here) are the failures model for CSP [BROS84] [TAUD90], and assertional s-rings [MAIM89].

2.1.2.2.3. Isomorphism. Technically, what is needed is an isomorphism between the abstraction semantics and non-determinism semantics. Such a mapping has been found for programs between predicate transformers and power domains [PLOG84] [APTK86], and another suggested for specifi-

cations [NELG89].

2.1.3. Wide- versus Narrow-Spectrum Languages. A major contribution of semantic models is the definition of abstraction orders. These orders prove useful in relating programs of various levels of abstraction (i.e. specification to design, and design to code). So they provide a means for relating levels of abstraction used in software development and maintenance.

These levels of abstraction are expressed in various languages, for example, the programming and specification languages discussed on page 5. Recall that the specification language is an extension of the programming language; consequently, the programming language is simply a subclass of specifications. The specification language is an example of a *wide-spectrum language*. A wide-spectrum language is one in which all levels of abstraction (from specification to code) can be expressed. One such language suffices for representation of the system throughout all development and maintenance activities [NEIJ84].

Typically, however, the languages used in software development are not intended to cover the full spectrum of abstraction. Instead, multiple languages are used, each intended to be used during a particular "phase" of development. These languages are called *narrow-spectrum* because they cover only a limited range of abstraction [NEIJ84]. The semantics of narrow-spectrum languages have not been well-studied. One possible exception is Back's derivation of a specification language, which combines two (non-equivalent) dual semantic structures to form the whole [BACR89]. This suggests a conjecture that the narrow-spectrum languages should correspond to complete sub-structures (sub-lattices) of a wide-spectrum language.

2.2. Programming.

The second division of the literature context is based on the concept of programming. This process of constructing and modifying programs is the primary force in software evolution. This section focuses on models, methodologies, and environments and tools for programming.

The models of programming discussed (including the more traditional life-cycle models and newer "process" models) serve as a description of the programming activities. Methodologies guide the programming activities and shape the process (described by models of programming). To help in the programming activities (especially in support of a methodology), environments and tools are used. Each of these aspects of programming is described below.

2.2.1. Models of Programming. The attempt to understand a complex idea is helped by finding

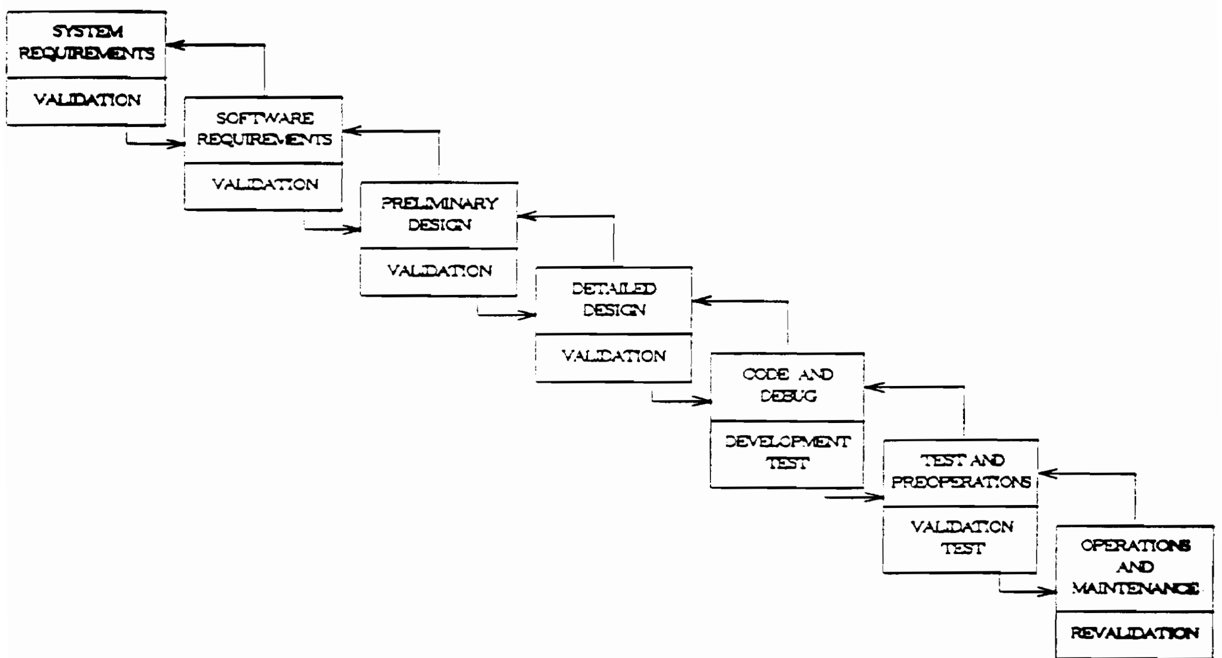


Figure 2.1. The Waterfall Life-Cycle Model.

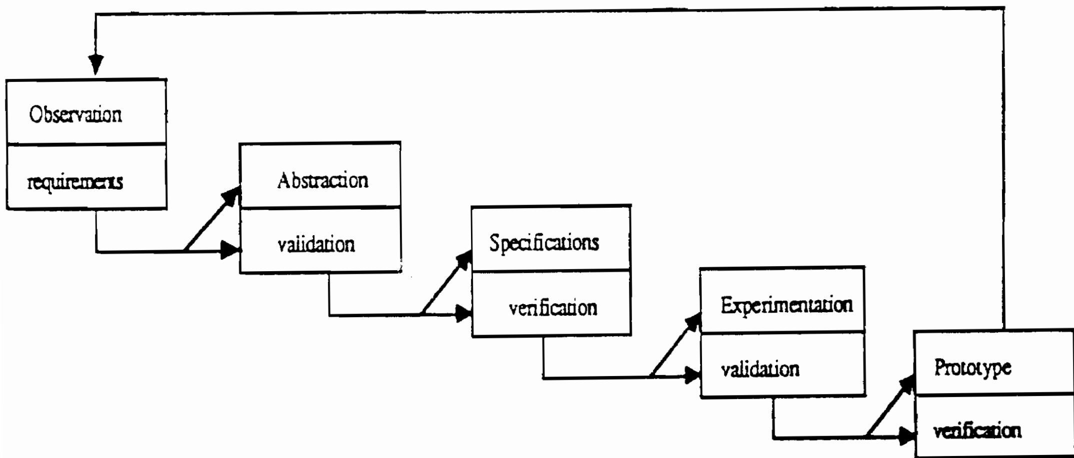


Figure 2.2. Giddings' Domain Dependent Software Life-Cycle Model.

a descriptive model, and programming is no exception. The complexity of programming has led to many different models. These can be classified into two types: life-cycle models, and process models. What is called programming here, is often called the "software process," with models described as "software process models" [DOWM86]. (These terms are not sufficiently descriptive, and could refer to many things. For example, "software process" could refer to the software evolution process, software development process or software maintenance process. Any of the above terms would lend clarity, however which is meant depends on the purpose of the model.)

2.2.1.1. Life-Cycle Models. Life-cycle models are the older form of programming models, with the original attributed to Bennington [BENH56]. The implication of "life-cycle" is that the models should include those activities which recur throughout the life of a software system. Technically, these models should make no mention of either development or maintenance, instead placing emphasis on those abstract activities which occur in both.

The most common deficiency of life-cycle models is the inability to represent maintenance and its connection to development. Oddly enough, under the interpretation of "life-cycle" given above, these issues cannot be addressed by such a model since the level of abstraction hides the details of both development and maintenance. Clearly this interpretation is not valid for most existing models.

Other deficiencies of life-cycle models include being acyclic, and showing methodological influences. No particular model is subject to all criticisms, nor do all criticisms apply to any specific model.

Consider the "classic" model: the Waterfall Life-Cycle Model as described by Boehm [BOEB76]. In this model the process "flows" down through several development stages into the "final" stage of maintenance (Figure 2.1). The subject of much well-deserved criticism, the waterfall model is not even cyclic. The deemphasis on maintenance, while outlining the development process in great detail, suggests a lack of understanding of what maintenance is and how it should be approached.

Improvements on the waterfall model also suffer. Giddings' Domain Dependent Software Life-Cycle Model [GIDR84] (Figure 2.2) and Boehm's Spiral Model [BOEB86] (Figure 2.3) are two examples. Both models are cyclic, and each corresponds adequately to the interpretation of life-cycle. However, both stress development activities instead of treating activities at a level encompassing both development and maintenance. The assumption that development activities are sufficient for maintenance represents the view that maintenance is a "miniature development cycle" as expressed by Peters [PETL89]. Although the traditional one, this view is recognized as being incorrect

[CHAN88].

Not all life-cycle models can be criticized in this manner. A primary example is the Re-Engineering Cycle of Bachman [BACC88] which emphasizes reverse engineering as part of the software evolution process. The model is split in half with forward and reverse engineering aspects. The forward engineering side is much like the waterfall model, leading from requirements to operational systems. The reverse engineering side allows the "reversal" of forward engineering, "lifting" and feedback of information from lower levels to higher ones (Figure 2.4). Unlike the other models, this model is oriented toward the complete life of software systems, with the aim of defining tools to support forward and reverse activities throughout the life of a system [BACC88].

The classic model of software evolution is the Two-Leg Model of Lehman et al. [LEHM84]. This model, whose name comes from its characteristic inverted "v" shape, shows the three dimensions of software evolution as sides of a triangle (Figure 2.5). These dimensions are abstraction, reification and the application/operational system "continuum." A development process takes an application concept (in the lower left-hand corner) and abstracts it to a formal specification (moving diagonally upward to the right). From the formal specification, reification (refinement) begins, transforming the specification through a variety of means to an operational system (in the lower right hand corner).

The two-leg model is difficult to understand without narrative, and its ability to depict software evolution is not obvious. The depiction of software evolution can be understood by noticing the mechanism of backtracking as part of reification. By backtracking to a previous realization of the application concept, new systems can be found by further reification. Software evolution is described as an alternating sequence of reification and backtracking steps.

A more recent life-cycle model showing characteristics of software evolution is the Process-Centered Software Life-Cycle Paradigm [GAMS88]. The "process" described is close to what is called a "methodology" in the next section. In this model the process (methodology) can be changed, and must be built before development of software. The model has three connected loops: the process static cycle, in which the process (methodology) is constructed; the product cycle, when the process (methodology) acts on the product; and the process/product dynamic cycle, in which either the process or product can be modified while still executing.

2.2.1.2. Process Models. As noted above, the life-cycle models are open to criticism for ignoring maintenance. More specifically, the criticism is that activities common to both maintenance and development should be represented in such a way that the linkages between the two can be portrayed.

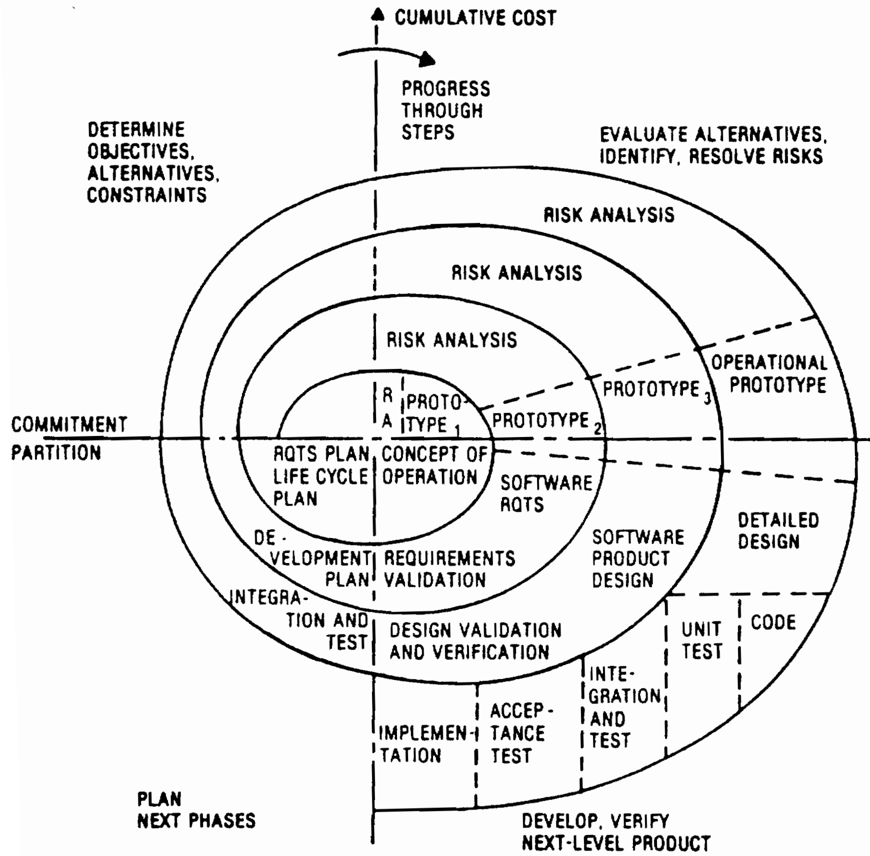


Figure 2.3. Boehm's Spiral Life-Cycle Model.

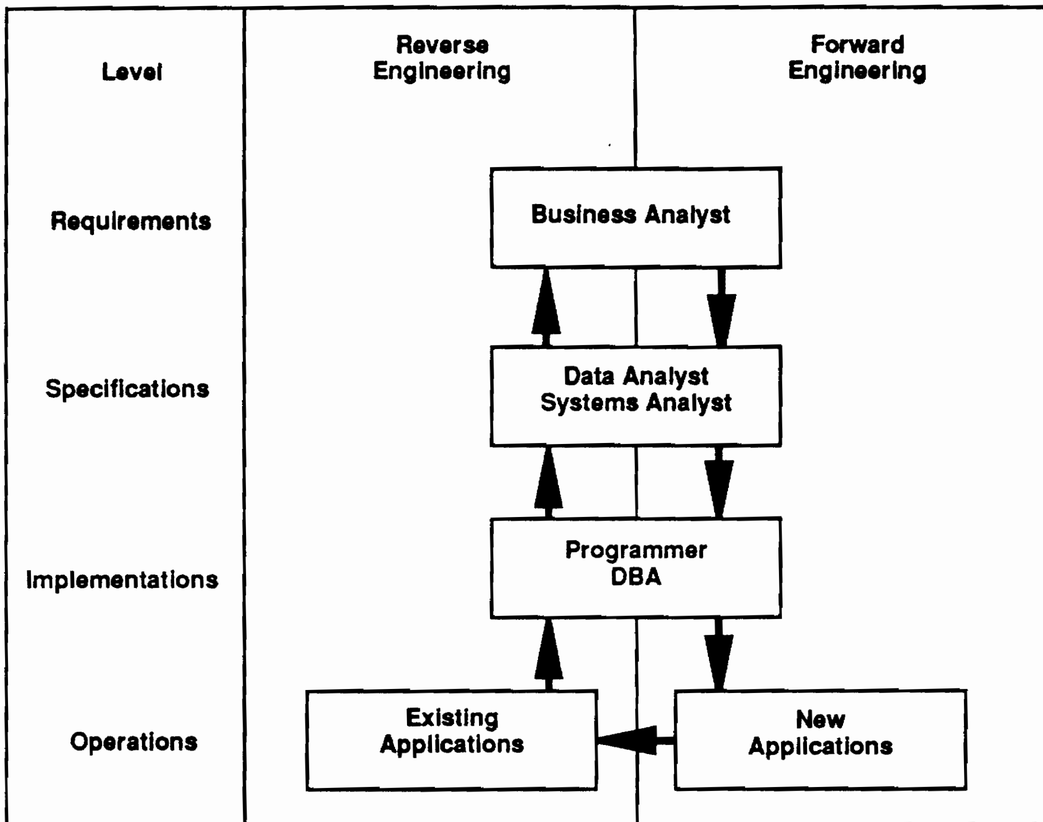


Figure 2.4. Re-Engineering Cycle (adapted from [BACC88]).

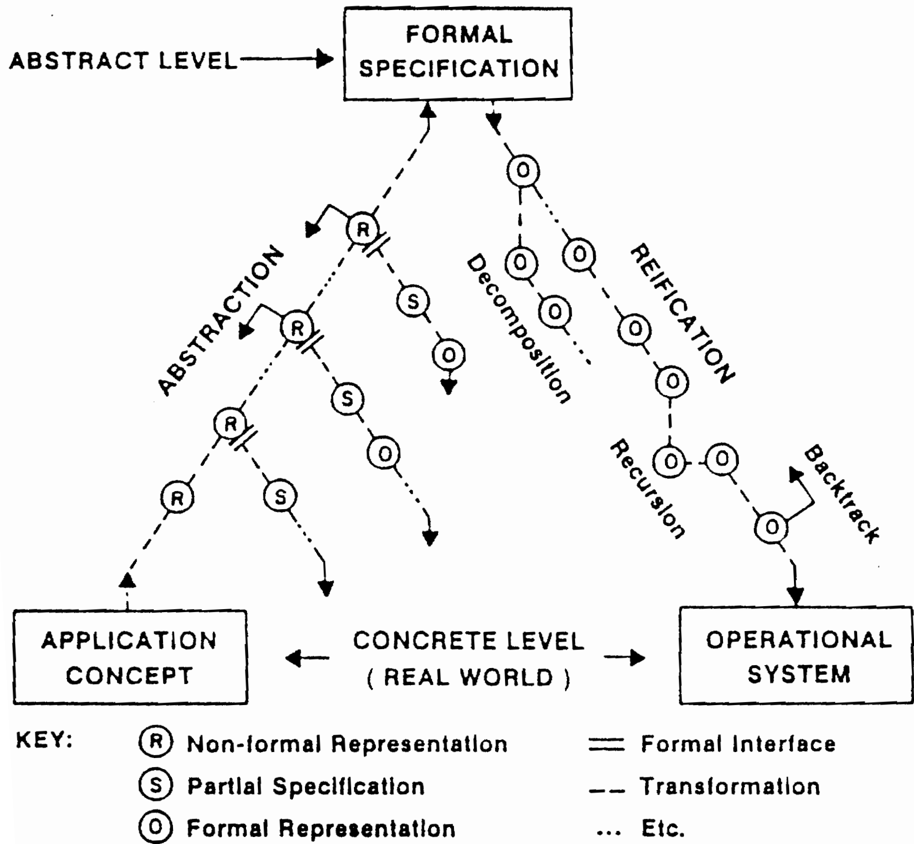


Figure 2.5. The Two-Leg Model (from [LEHM84]).

By emphasizing development and methodological views, these representations prevent visualization of the evolution process.

Models of the software evolution process are not open to similar criticism. "Process" models avoid these pitfalls (ignoring maintenance, and methodological proscription) by representing only the "basic" activities of development and maintenance. The Abstraction Refinement Model is a software evolution process model, and its utility and efficacy should be judged in comparison with these models.

Process models try to reduce the inherent complexity of the software evolution process by considering only how basic activities interact. More complex activities can then be constructed from the basic ones.

Four different approaches are presented: the Integrated Life-Cycle Model of Yau [YAUS84] [YAUS88], the Software Project Model of Baker et al. [BAKA86], the Draco Paradigm of Neighbors [NEIJ80] as extended by Arango et al. [ARAG85], and a model defined by Bergstra [BERJ84]. These models use the system descriptions with operations on them as a basis. The primary distinction is in representation: while the first three are graph based, the last is algebraic.

2.2.1.2.1. Integrated Life-Cycle Model. The Integrated Life-Cycle Model is intended to serve as a history of development. A graph-based language is used to represent the system with abstract notation for semantic notions of "Control flow, data flow, and data structure." Each "phase" of a major activity (such as specification or high- level design) is represented by a graph of the system, with "interphase" activities (really those of the next phase) represented by applications of graph-rewriting rules [YAUS84] [YAUS88]. Thus, the model is a sequence of system descriptions with the definition of the transitions between them.

2.2.1.2.2. Software Project Model. A graph-based representation is also used in the Software Project Model. Each such model is a collection of "document histories" which are acyclic graphs showing the evolution of a particular document through different versions. The document version includes the document itself and a time-stamp, thus allowing the representation of the system description (collection of documentation) at any particular time [BAKA86]. One deficiency of this model is the lack of an explicit "derivation" relationship between documents although the existence of abstraction orders is indicated (in the form of a "correctness with respect to" relation [BAKA86]).

2.2.1.2.3. Draco Paradigm. The third graph-based process model is the Draco Paradigm. This

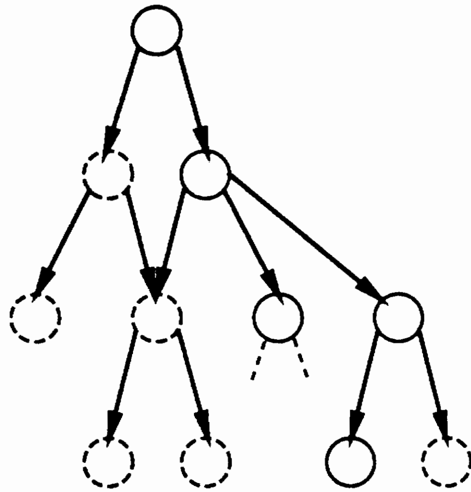


Figure 2.6. The Domain Structure Graph of Draco (adapted from [ARAG85]).

paradigm is the extension of the work of Neighbors [NEIJ80] in an approach to software development designed to promote the reuse of analysis and design. The fundamental idea behind Draco is that software must be developed "within" the application domain. In this way the system is represented in terms of the context of the domain.

The formal model used by Neighbors is based on (transitional) petri-nets [NEIJ80]. A petri-net is a graph consisting of two types of nodes, places and transitions, and arcs between them. The places are used to represent components of the application domain, and the transitions, refinements on these components. A marking of the petri-net indicates the information particular to a software system (or a problem for which the system is a solution). Transitions fire only when complete information is available to make the refinement. In this way, not only are the system components described, but also the refinements and their firing conditions as well.

The Draco Paradigm models the process in a slightly different manner. Using Neighbors notion of refining domains from high-level specifications to executable ones [NEIJ80], Arango et al. define the *domain structure graph* [ARAG85]. A domain structure graph is a directed acyclic graph in which the nodes are the domains and the arcs are domain refinements or design decisions (Figure 2.6). The leaves of the graph are executable specifications while the root is a high-level specification. The development process is described by downward traversal of a path of arcs, and maintenance by first "reversing" the arcs and then traversing downward to a new realization [ARAG85]. This paradigm greatly influences the Abstraction Refinement Model (especially the original conception, see [NANR89]).

2.2.1.2.4. Bergstra's Model. The fourth process model is from Bergstra and is algebraic. This model is similar to that of the Draco Paradigm because a theory analogous to a domain is used to describe a class of systems. Corresponding to this theory are algebras which represent the levels of abstraction. These levels of abstraction are combined into a hierarchy which consists of the algebras and homomorphisms which relate them.

The algebras representing the levels of abstraction have as elements, system descriptions, and as operations, tools for system construction. A set of basic systems, from which other systems can be constructed, are generators for the algebra. The homomorphisms between each pair of levels allow realization of each system in terms of the next lower level [BERJ84].

An interesting aspect of this model with respect to the intent for the Abstraction Refinement Model is the way in which algebraic representations are used. Bergstra's model represents the

abstraction levels algebraically, and seems intentionally to ignore abstraction orders. The ARM, on the other hand, "algebraizes" the abstraction order. Bergstra's model might be called *horizontal* and the ARM *vertical*, or alternatively *layered* and *homogeneous* to stress the distinction.

2.2.2. Methodologies. The models in the last section are attempts to describe the process through which software evolves. Methodologies attempt to govern this process. In the subsequent discussion, emphasis is placed on methodologies which include explicit descriptions of the process. The treatment is divided between development and maintenance.

The term "methodology" is frequently interpreted in many ways, and is often confused with the term "method." Following the lead of Freeman [FREP77], a distinction is made between the two. A *method* attacks one task, defining the decisions needed to be made, the means for resolving them and the order in which they are made. A *methodology* is a collection of complementary methods with rules for their application. Further understanding of the concept of a methodology can be gained from the work on the Objectives, Principles, and Attributes framework [ARTJ86].

2.2.2.1. Development. The methodologies for software development are abundant. Those which capture the process (have a corresponding process model); however, are not. The primary examples considered are programming calculi which originated with Dijkstra [DIJE76] and include "large" methodologies such as VDM [JONC80]. Rather than consider any one methodology based on programming calculi, the methodological contributions of such calculi are reviewed.

A programming calculi consists of a formal notation for specification, and a set of design rules. Three formal notations for functional specifications are assertional, functional and relational [MILA86]. Each allows the specification of the input/output behavior of the system. The assertional notation is used for the comparison of programs and specification in Chapter 3.

Design rules serve a role similar to the transformations described in the Integrated Life-Cycle Model and the Draco Paradigm. These rules modify the specifications so they are transformed into correct realizations. Two such rules are the assignment rule and decomposition rule. The first of these maps a "simple" specification into an assignment statement, and the second decomposes the specification into several simpler specifications [MILA86]. These rules and others, are applied during development to transform a specification into an executable realization which satisfies the specification.

2.2.2.2. Maintenance. Methodologies for maintenance are rare, despite the recognized need

for harnessing maintenance problems [NANR90]. Two methodologies are the Draco Paradigm [ARAG85], and that described by Yau [YAUS84]. Both these methodologies are closely tied to process models described in the previous section. Neither methodology is strictly for maintenance; the utility of each in maintenance is bolstered by its prior use in development.

2.2.2.2.1. Draco Paradigm. The Draco Paradigm is mentioned earlier as contributing the domain structure graph as a model of software evolution. The Paradigm is the methodology which corresponds to this model. As a result of the application of the methodology, a path through the domain structure graph is elaborated. The Paradigm initiates with an analysis of the problem area (or application domain) called *domain analysis*. This establishes a framework of application domain knowledge underlying the design of a domain language facilitating the description of systems. Draco provides an environment consisting of tools for transforming and refining the application domain into programming domains where implementations are possible. Domain analysis provides the means for determining what transformations are acceptable as part of the definition of the domain language.

These refining transformations are used primarily in development, where the original domain is mapped into a programming domain. This corresponds to the elaboration of a path from the root to a leaf of the domain structure graph (Figure 2.7). This path provides a record of the series of design decisions through which the system is constructed. Maintenance can then be achieved by reversing the design decisions from the implementation toward the root until a common abstraction of the current and desired implementations is achieved (Figure 2.8). The inverse refinement process works well for a system constructed using Draco since the design decisions have been saved through application of the paradigm. For a system implemented by other means it is not necessarily the case that those decisions are available for use in maintenance.

A solution to this problem is provided by a method for *design recovery* for which the implementation serves as a source of information. This information must be augmented by knowledge of the application domain in order to reverse the effects of the refinements used in development. By combining this information the domain structure graph is constructed "bottom up" and is available for maintenance [ARAG85].

2.2.2.2.2. Methodology of Yau. The use of a model of the development of a software system is a characteristic shared by the methodology described by Yau [YAUS84]. This model (the Integrated Life-Cycle Model described earlier) shows the derivation and relationships among the products of the various phases of development. The model, like that of Draco, could either be a product of

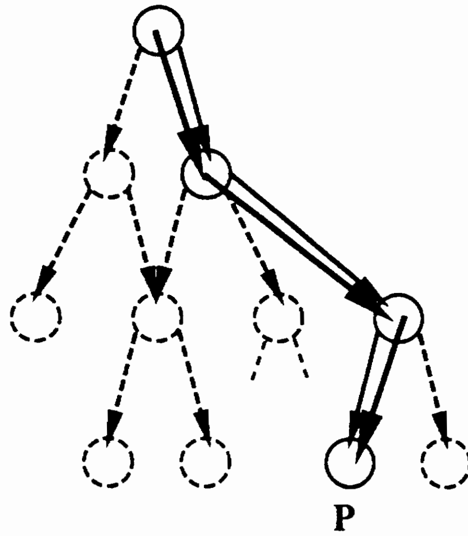


Figure 2.7. Draco Characterization of Development.

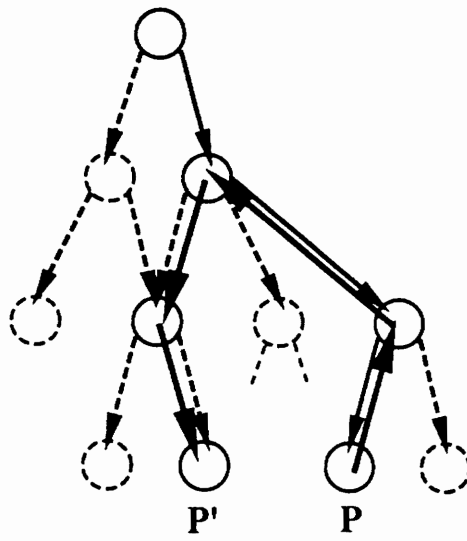


Figure 2.8. Draco Characterization of Maintenance.

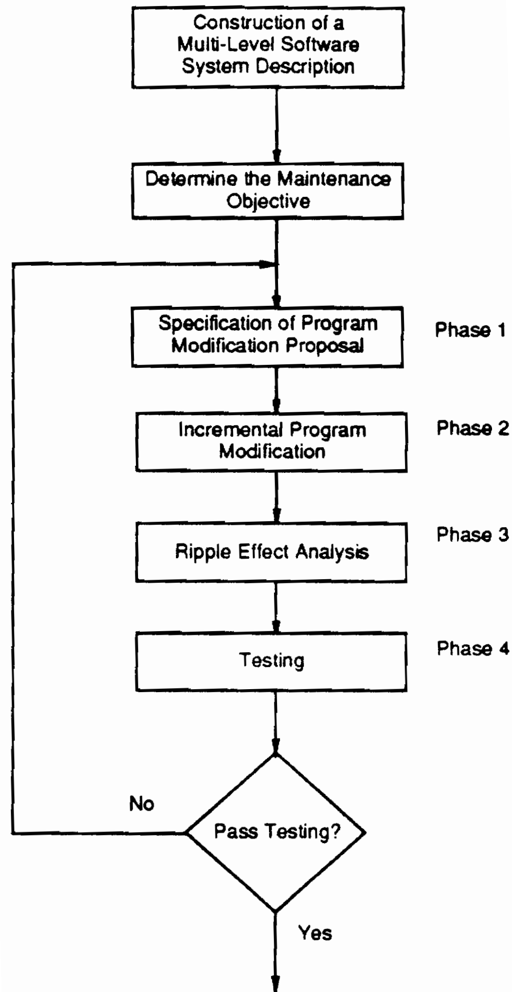


Figure 2.9. Yau's Maintenance Methodology.

development or of a pre-maintenance effort, and is updated with the code during any maintenance activity.

The remainder of the methodology sequences the tasks of implementing a particular change (see Figure 2.9) following the determination of the immediate maintenance objective. An iterative process of solution proposal, implementation, analysis and testing completes the modification process. The methodology provides methods for performing each of the individual tasks to ensure reliable modification of the existing system. The underlying model assists in this process by providing both information and a means for studying proposed modifications.

2.2.3. Environments and Tools. The complex methodologies for software evolution need computerized tools to implement the methods. The set of integrated tools that support a methodology is called an *environment*. Both the Draco paradigm and the maintenance methodology of Yau have corresponding environments.

One important tool used in both methodologies is a reverse engineering scheme for design recovery. A similar tool is called the Information Abstractor [ANTP87]. The purpose of the tool is to generate documentation that can replace existing unreliable or non-existent documentation. The underlying model of the tool is similar to the Re-Engineering Cycle of Bachman described above. Development occurs through a "Direct Engineering" process, with feedback connections to a "Reverse Engineering" process. The documentation is generated by a "Virtual Information Abstractor" which encompasses the reverse engineering process.

More formal approaches to defining programming environments are based on the idea that programming must be well formulated as an activity before a supportive environment can be defined [BROM86]. Most of this work has been approached by using Algebraic Specification (based on Abstract Data Types) in combination with semantically valid transformations [KRIB88], [KRIB89], [BROM86]. An example designed to support development of systems in Ada is PROSPECTRA [KRIB88] [KRIB89]. In contrast with the view taken in this research, Krieg-Brückner considers maintenance as redevelopment using the "development script" following modification of the requirements [KRIB89, p.37].

These environments could easily be augmented by the addition of tools for making reverse transformations. Such tools are part of the environment supporting the Draco Paradigm, and recent research efforts have addressed them [SNEH88] [OVEC88] [KELS88]. These reverse transformations form an important part of the contribution of Draco to the ARM.

2.3. Program Execution.

The process of programming is intended to describe (or specify), via a program, a desired behavior of a computer system in the real world. This behavior is termed *program execution*, which is characterized by program operations in performing specified computational tasks.

The emphasis on "real world" computational machines is important. This is the machinery on which the software is executed, and is to be differentiated from computational models. Most computational models are well-defined mathematically, and the (operational) semantics of programs are fairly concisely stated. The idiosyncrasies and uncertainties of the electronic components of a computer may affect the behavior (with respect to a computational model) in subtle ways, preventing formal and real-world modes of behavior from being identical.

This discrepancy between the real-world computers and models of computers should raise some concern over the use of formal techniques (see [FETJ88]). However, this separation is a common facet of applied mathematics where many real-world situations are modeled [BARJ89]. If the computer and computational model are nearly isomorphic, the uncertain behavior of the computer can be ignored (except for certain real-time systems). The crucial issue for this research is to realize that it deals with both computational models as a basis and software evolution models as a result, and neither have the exact behavior of the real-world entities they model.

One aspect of the real world that it is important to include in the model is the occurrence and meaning of "bad" software. A *degenerate* program contains one or more *faults* which specify *failures* in the program's behavior. In some sense a fault is a deviation from a perfect program which satisfies the initial specification of the system. However, specifications describe a desired behavior which may be shared by many perfect programs [MUSJ87]. It is better to think of a fault as a specification of a failure which is a deviation from the desired behavior of the system (i.e. the notion of fault is closely tied to the semantics). This view of a fault is more realistic since a program never executed appears to be correct despite any failures which could occur if it were executed.

2.4. Summary.

This chapter has covered a diverse number of topics under the framework of "program," "programming" and "program execution." Each of these subjects influences the definition of a software evolution model in some way either by contributing a concept or placing a requirement on the process. The diversity indicates the potential universality of such a model.

The abundance of topics forming the literature context is also indicative of the generality that the model must encompass. Fortunately, many common themes occur in the literature and motivate the definition of the ARM. These are:

(1) Development is a series of refining transformations.

(2) Abstraction orders can be defined to model refinement.

(3) Reverse engineering serves an important role in maintenance.

(4) Reverse engineering can be modeled as reverse transformations.

Chapter 3. Foundations

The development of the Abstraction Refinement Model in Chapter 4 draws heavily on the literature presented and on mathematical concepts. The foundations of this development are elaborated below.

The mathematical preliminaries are presented first. These include concepts from lattice theory which are used to describe the structure of the ARM. In addition, the Lindenbaum algebra of a first-order logic is discussed. The latter is used as a basis for the construction of the intended semantics of the ARM in Chapter 4.

Following the mathematical preliminaries, the semantic and syntactic foundations are presented. For the semantics, an axiomatic model is presented, and for the syntax, the Laws of Programming are discussed. Each of these sections provide foundations for the work presented in Chapter 4.

3.1. Mathematical Preliminaries.

The material presented in this and the following chapters is based on concepts from lattice theory. The appropriate elements are presented in the following. Additionally, the Lindenbaum algebra is discussed since it is used for a more precise development of the structure of the intended semantics in Chapter 4.

3.1.1. Lattice Theory. The approaches in this work require knowledge of partially ordered sets and lattices. An introduction to these concepts follows. The discussion here is based on the book by Grätzer [GRÄG78], but another popular source is Birkhoff [BIRG40]. The important notions are based on relations.

A *binary relation* on the sets A and B is a subset of their cross product $A \times B$. If R is a relation $R \subseteq A \times B$, then $a \in A$ *relates* to $b \in B$ (denoted aRb) if $(a, b) \in R$. The relations essential here are *orders*, which are generally binary relations on one set A. An order is typically denoted by a symbol like $\leq$, and the symbols used here include $\leq$, and $\sqsubseteq$.

Different types of orders are distinguished by four properties. Let $\leq$ be an order relation on A and for $a, b \in A$:

(1) (Reflexivity) $a \leq a$.

(2) (Transitivity) $a \leq b$ and $b \leq c$ imply $a \leq c$.

(3) (Anti-symmetry) $a \leq b$ and $b \leq a$ imply $a = b$.

(4) (Linearity) $a \leq b$ or $b \leq a$.

An order having properties (1) and (2) is called a *pre-order*. By adding property (3), a pre-order becomes a *partial order*, and an order with all properties is a *linear order*.

An order is not used without reference to a set, and usually a non-empty set is referred to as being ordered and is denoted $\langle A, \leq \rangle$ (most of the time denoted only by A). A ***pre-ordered set*** has $\leq$ as a pre-order, also a ***partially ordered set*** (a ***poset***) has $\leq$ as a partial order. A poset $\langle A', \leq \rangle$ can always be constructed from a pre-ordered set $\langle A, \leq \rangle$ by finding the ***quotient*** of A with respect to the equivalence relation based on $\leq$. (An equivalence relation is a binary relation which is reflexive, transitive and symmetric. Symmetry is the property that for all $a, b \in A$, aRb implies bRa.)

This equivalence relation $\equiv$ is defined by $a \equiv b$ *iff* $a \leq b$ and $b \leq a$. That this is an equivalence relation is easy to see. Reflexivity and transitivity follow from $\leq$ being a pre-order. Symmetry comes from the definition, and the fact that "and" is commutative (i.e. we can swap $a \leq b$ and $b \leq a$). The poset $\langle A/_\equiv, \leq/_\equiv \rangle$ consists of the set of equivalence classes of A, and the order on the equivalence classes $[a] = \{b \in A \mid b \equiv a\}$:

$$A/_\equiv = \{[a] \mid a \in A\}$$

$$[a] \leq/_\equiv [b] \quad \textit{iff} \quad \forall x \in [a] \text{ and } \forall y \in [b], x \leq y.$$

The anti-symmetry of $\leq/_\equiv$ can be seen by noting that if $[a] \leq/_\equiv [b]$ and $[b] \leq/_\equiv [a]$ then for all $x \in [a]$ and $y \in [b]$, $x \equiv y$.

A linearly ordered subset of a pre-ordered set or a poset is called a ***chain***. Any elements $a, b \in A$ such that $a \not\leq b$ and $b \not\leq a$ are called ***incomparable***. A subset of incomparable elements of A is called an ***antichain***.

The ***upper bound*** of a subset X of A is an element $x \in X$ such that for all $y \in X$, $y \leq x$. A ***least upper bound*** or ***supremum*** of X is an $x \in X$ such that for all upper bounds b of X, $x \leq b$. The uniqueness of the least upper bound is only guaranteed if antisymmetry holds, in which case the least upper bound of X can be denoted $\bigvee X$ or $\sup X$. If $X = \emptyset$ (the empty set) then $\sup X$ is an element which is the upper bound of no other element. So if A has a least element then $\sup \emptyset$ is defined and denotes this least element (called the *zero*, 0, or *bottom*, $\perp$).

A similar definition can be given for the lower bounds. The greatest lower bound or infimum of $X \subseteq A$ is an $x \in X$ such that $b \leq x$ for all lower bounds b of X, and is denoted $\bigwedge X$ or $\inf X$. Again,

the uniqueness of the greatest lower bound is only guaranteed by antisymmetry. The greatest lower bound of the empty set only exists if there is a greatest element of A (called the ***unit***, 1, or ***top***, $\top$). (Notice that the definitions of supremum and infimum are ***dual***; meaning only the orders $\leq$ need to be "flipped" to find the other. That any statement about posets has a dual found in this manner is called the ***duality principle***.)

A partially ordered set $\langle A, \leq \rangle$ is a *lattice* whenever both $\sup X$ and $\inf X$ exist for any non-empty set X of A. A *complete lattice* is one for which all subsets have a supremum and infimum. Two operators, join $\vee$ and meet $\wedge$, can be defined in terms of the supremum and infimum of the elements a and b:

$$a \vee b = \sup \{a, b\}$$

$$a \wedge b = \inf \{a, b\}$$

It can be shown that $\vee$ and $\wedge$ relate to the ordering as follows:

$$a \leq b \quad \textit{iff} \quad a \wedge b = a$$

$$a \leq b \quad \textit{iff} \quad a \vee b = b$$

Using these definitions the following properties can be shown for meet and join.

(1) (Idempotency)	$a \wedge a = a$	$a \vee a = a$
(2) (Commutativity)	$a \wedge b = b \wedge a$	$a \vee b = b \vee a$
(3) (Associativity)	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$	$a \vee (b \vee c) = (a \vee b) \vee c$
(4) (Absorptivity)	$a \wedge (b \vee a) = a$	$a \vee (b \wedge a) = a$

An algebra $\langle L, \wedge, \vee \rangle$ for which these identities hold is also a lattice and can be shown to be "equivalent" to the poset notion. A *Boolean lattice* is a complemented, distributive lattice:

(4) (Distributivity)	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
(5) (Complementation) There exists an $\bar{a} \in L$ such that	$a \vee \bar{a} = \top$	$a \wedge \bar{a} = \bot$

A ***Boolean algebra*** is the corresponding algebra with 0, 1 and the operation of complementation ($'$): $\left\langle B, \wedge, \vee, ', 0, 1 \right\rangle$ [GRÄG78].

In addition to these definitions, orders regarding functions on ordered sets are needed. Given two sets A and B, with B ordered by $\leq_B$, then the set of functions from A to B (denoted $A \rightarrow B$) can be ordered by

$$f \sqsubseteq g \quad \textit{iff} \quad f(x) \leq_B g(x) \text{ for all } x \in B$$

The properties that hold for $\langle B, \leq_B \rangle$ also hold for $\langle A \rightarrow B, \sqsubseteq \rangle$. A function $f : A \rightarrow B$ is ***monotonic*** (or ***isotone*** [GRÄG78]) if A is ordered by $\leq_A$ and $x \leq_A y \implies f(x) \leq_B f(y)$ for all $x, y \in B$. The set of monotone functions from A to B is denoted $\lceil A \rightarrow B \rceil$ [MORJ89].

3.1.2. Lindenbaum Algebra. The concepts of lattice theory are used in the definition of Lindenbaum algebras. The Lindenbaum algebra of a first order language is useful in the study of logical model theory [BELJ77]. In Chapter 4 it is used to order the set of predicate transformers. The purpose is to construct a Boolean algebra from the set of formulas.

When considering logical formulas it becomes obvious that non-identical formulas can be logically equivalent. In model theory and predicate transformer semantics only the "semantic" entities are of interest. To find these entities, the equivalence relation

$$\phi \approx \psi \quad \textit{iff} \quad \Sigma \vdash \phi \leftrightarrow \psi \qquad \text{for } \phi, \psi \in \Phi$$

is defined. (Σ is a subset of Φ, and might be the axioms of a theory.) Using this equivalence, the semantic entities are the equivalence classes

$$|\phi| = \{\psi \in \Phi | \, \phi \approx \psi\} \qquad \text{for } \phi \in \Phi.$$

The Lindenbaum algebra of L on Σ is formed on the set of equivalence classes

$$B = \{|\phi| \mid \phi \in \Phi\}$$

ordered by

$$|\phi| \leq |\psi| \quad \textit{iff} \quad \Sigma \vdash \phi \rightarrow \psi.$$

The order induces a Boolean algebra on B with join and meet defined for $\phi, \psi \in \Phi$ as

$$|\phi| \vee |\psi| = |\phi \vee \psi|$$

$$|\phi| \wedge |\psi| = |\phi \wedge \psi|$$

and the zero and unit as

$$\Sigma \vdash \phi \quad \textit{iff} \quad |\phi| = 1$$

$$\Sigma \vdash \neg\phi \quad \textit{iff} \quad |\phi| = 0.$$

The complement operator is $|\phi|^* = |\neg\phi|$.

These definitions are used to define the structure on predicate transformers. The *Lindenbaum algebra of L on Σ* is denoted $B(\Sigma)$, and if $\Sigma = \emptyset$ it is called simply the *Lindenbaum algebra of L* and denoted $B(L)$ [BELJ77].

3.2. Semantic Foundations.

Semantic models are used in the development of the ARM to substantiate the syntactic structure of the ARM. The model used for this purpose is discussed below.

The models of abstraction are based on predicate transformers as discussed by Dijkstra [DIJE76] [DIJE90]. In particular, these predicate transformers are those defined by the weakest and weakest liberal pre-conditions of specifications and programs. The relationship of these predicate transformers to the logical order of implication allows the construction of orders on programs ([MORC87] [BACR89] [MORJ89] [NELG89]) like the abstraction order of the ARM.

3.2.1. Predicate Transformers. Axiomatic semantics are based on functions on predicates called predicate transformers. The predicates are logical expressions which indicate the allowed values of variables and are written in a first-order language L (see [BELJ77] for a statement of first-order syntax). The variables of L are assumed to be in an enumerated list $v_0, v_1, v_2, \ldots$. The set of all formulas in L is denoted Φ.

Examples of predicates, or formulas, are $v = 0$, $(0 \leq x \leq 100)$. These place explicit restrictions on the values of the variables v and x: v must contain 0 and x must contain a value between 0 and 100. Also common in the literature are predicates *true* and *false*. The condition *true* indicates that any value assignment is legitimate, but *false* indicates that no value assignment can satisfy it. In formal logic the semantic values of truth and falsehood are not included in a logic system. However, because of their common usage, *true* and *false* are interpreted here to be "names" for arbitrary tautologies and contradictions in L.

A *predicate transformer* can be defined as a mapping $f : \Phi \rightarrow \Phi$, and a monotonic predicate transformer one that preserves implications:

$$\text{If } \vdash \phi \rightarrow \psi \text{ then } \vdash f(\phi) \rightarrow f(\psi) \qquad \forall \phi, \psi \in \Phi.$$

The *conjugate* of a predicate transformer f is f^* which is defined by $f^*(\chi) = \neg f(\neg\chi)$ [DIJE90]. Only monotonic predicate transformers are used below, because most predicate transformers corresponding to programs are monotonic [DIJE90].

3.2.1.1. Weakest Pre-Conditions. The predicate transformers used to define the semantics of programs are considered mappings from post-to pre-conditions. These conditions are predicates which indicate the permitted initial values of the program variables. The reverse nature of the mapping allows the representation of programs by total functions without sacrificing non-termination.

Semantic descriptions are given in terms of weakest pre-conditions and weakest liberal pre-conditions. The weakest pre-condition is the condition which allows the greatest number of assignments of values to the variables (typically called *states*) such that the program terminates and the post-condition is satisfied [DIJE76] (Dijkstra does not explicitly discuss programs only "mechanisms" and "happenings," see [PLOG80] for a clarification). The weakest liberal pre-condition similarly allows the greatest number of assignments of values such that if the program terminates the post-condition is satisfied.

The notation for weakest pre-conditions varies. Dijkstra uses $wp(p, \phi)$ [DIJE76], and more recently $wp.p.\phi$ [DIJE90], for the weakest precondition of program p with post-condition ϕ. Others ([BACR89] [MORJ89] [MORC88]) simply denote it by some variation on $p\langle\phi\rangle$. The notation for weakest liberal pre-condition is similar.

Both wp and wlp are required to totally characterize the behavior of programs. Because of this, a single semantic entity, a predicate transformer, cannot be considered to correspond to a program.

The need for both predicate transformers $wp.p$ and $wlp.p$ to make sense of the behavior of a program p can be seen in the definitions given by Dijkstra [DIJE90]. The result $wp.p.true$ is a predicate which holds in any initial state from which all computations of p terminate. The result $wlp.p.\phi$ is a predicate which holds in any initial state from which each computation either terminates satisfying ϕ or does not terminate. The predicate transformer $wp.p$ is defined as the conjunctions of $wp.p.true$ and $wlp.p.\phi$. So $wp.p.\phi$ holds only in any initial state from which each computation terminates satisfying ϕ.

These definitions enable representation of different behaviors by each predicate transformer. The predicate transformer $wlp.p$ specifies the initial states which may lead to acceptable forms of termination, and $wp.p.true$ indicates initial states from which the program terminates. Together they give the initial states from which only acceptable termination occurs. The complete behavior

of a program cannot be described without both *wp* and *wlp*, and they cannot be combined to form a single meaningful predicate transformer. So the semantics of a program in terms of *wp* and *wlp* are given as triples showing $wlp.p.\phi$, $wp.p.true$ and $wp.p.\phi$ (although the last can be derived from the first two).

3.2.1.2. Properties. Certain properties of predicate transformers are used to describe programs and to differentiate them from specifications. The properties discussed here are monotonicity, determinism, feasibility (the Law of the Excluded Miracle), conjunctivity and continuity. Dijkstra has restated these properties in more general terms for predicate transformers not defined with either *wp* or *wlp* [DIJE90]. The restated forms may provide a better formulation of the requirements to be met by a programming language than the following. However, this improvement has not been applied to differentiating programs from specifications, and is not discussed here.

The property of monotonicity is stated above. It proves particularly useful in the definition of recursion (see for example [NELG89]). Monotonicity of predicate transformers means that if one predicate implies another, then the result for the first must imply the result for the second. Implication is like a "weaker" relation: $\phi \rightarrow \psi$ is a statement that "ψ is weaker than ϕ." In this sense, weaker means allows more states or value assignments. So for two predicates, the fact $\phi \rightarrow \psi$ means that for every state satisfying ϕ it also satisfies ψ. Transferring this to pre-conditions the statement says every initial state of p satisfying $wp.p.\phi$ is also an initial state satisfying $wp.p.\psi$.

What monotonicity enforces is a "consistency" property. All it says is that a weakening of the post-condition results in a weakening of the pre-conditions. Meaning for a program that if the set of desired outputs is enlarged, then no input states are lost (only gained). This is a property that is held by programs, and the assumption reflects that fact.

The determinism of programs can also be characterized by the predicate transformers. By playing with the definition of *wlp*, it can be seen that $\neg wlp.p.(\neg\phi)$ holds in all initial states from which there exists a terminating computation such that ϕ is satisfied. Obviously, $wp.p.\phi \rightarrow \neg wlp.p.(\neg\phi)$ for any program p. But for a deterministic p, if a terminating state satisfying ϕ is possible, then it is the only one. So a program p is deterministic whenever $wp.p$ and $wlp.p$ are conjugates of each other [DIJE90].

Programs and specifications are differentiated by the remaining properties. Specifications are defined by extending the syntax of the programming language to include "prescriptions." Prescriptions are pre- and post-condition pairs that specify a "mechanism" which satisfies the post-condition

if the mechanism initiates in the pre-condition [MORJ89]. The extension of the syntax also requires relaxation of the properties of programs. So not only is the specification language a super-class of the programming language, it is also more general [MORJ89] [BACR89]. The inclusion of prescriptions results in the loss of some of Dijkstra's "healthiness conditions" [BACR89]. The complete set of healthiness conditions are feasibility, conjunctivity and continuity. All of which apply to programs.

PROPERTY 3.1 (FEASIBILITY). *For a program p,* $wp.p.false = false$

The property states that no program can terminate with a variable assignment such that $false$ is true. A program violating this property is called a miracle since it would have to create an assignment to program variables which satisfies $false$, which is impossible. (Back [BACR89] suggests that the Law of the Excluded Miracle is a strictness property, however strictness relates to non-termination which has no relation to $false$. The term "feasibility" comes from Morgan [MORC88].)

PROPERTY 3.2 (CONJUNCTIVITY). *For any program p and post-conditions* ϕ *and* ψ ($wp.p.\phi$ and $wp.p.\psi$) *iff* $wp.p.(\phi \wedge \psi)$.

PROPERTY 3.3 (DISJUNCTIVITY). *For any program p and post-conditions* ϕ *and* ψ ($wp.p.\phi$ or $wp.p.\psi$) $\Longrightarrow$ $wp.p.(\phi \vee \psi)$.

For deterministic programs equivalence can be shown.

PROPERTY 3.4 (DISJUNCTIVITY). *For any deterministic program p and post-conditions* ϕ *and* ψ ($wp.p.\phi$ or $wp.p.\psi$) *iff* $wp.p.(\phi \vee \psi)$.

PROPERTY 3.5 (OR-CONTINUITY). *If* $\phi_0, \phi_1, \phi_2 \ldots$ *is a monotonic sequence of predicates, and p any program, then for all states* $wp.p.(\exists r(\phi_r)) = \exists s(wp.p.\phi_s)$.

(A monotonic sequence is one for which $\phi_i \Longrightarrow \phi_{i+1}$ for $i \geq 0$, over all states.) Or-continuity forbids unbounded non-determinacy in programs. Unbounded non-determinacy is the property such that a program is guaranteed correct termination on some fixed input, however the set of possible responses is infinite for any given input. Although the only feasible form of non-determinism, the desirability of bounded non-determinism has been questioned for concurrent programs [CLIW81].

For specifications the properties of continuity and feasibility are not generally valid. Continuity is violated by the prescription $chance\ (true \,||\, true)$ under infinite states. Feasibility is violated by the

prescription $(\mathit{true} \,\|\, \mathit{false})$. Other prescriptions also violate feasibility and are called either *miracles* [BACR89] [MORC88] [MORJ89] or *partial commands* [NELG89]. Miracles are sub-implementable (in the sense of the abstraction order described later), and correspond to the observation that specifications need not be implementable.

The operational interpretation of miracles is difficult to comprehend. Simply stated, they are programs which cannot be initiated in certain states. A good example is "naked" guarded commands: $B \rightarrow P$. As noted by Morgan [MORC88], all prescriptions have corresponding guarded commands. Miracles are simply those which do not allow initiation; consider $(\mathit{true} \,\|\, \mathit{false})$ which corresponds to $\mathit{false} \rightarrow P$ for any program P. (Note that one way to deal with miracles is via backtracking [MORC88] [NELG89].) The potential benefits of the use of miracles is explored by Morgan [MORC88].

3.2.2. Sample language. To illustrate the intended semantics of the ARM it is necessary to explore some of the prescriptions of a specification language. The prescriptions presented include *miracle*, *choose*, *skip*, *abort*, and *havoc*.

The first prescription considered is *miracle*, which has been discussed thoroughly. It is defined as follows

$$wlp.miracle.\chi = true \qquad \forall \chi \in \Phi$$

$$wp.miracle.true = true$$

$$wp.miracle.\chi = true \qquad \forall \chi \in \Phi$$

Together these indicate that no matter what state it is initiated in, *miracle* will achieve all outputs which may be desired. The *miracle*, of course, is that it can achieve no state from every state (i.e. $wp.miracle.false = true$).

Next is *choose* [BACR89] which is defined by

$$wlp.choose.\chi = (\exists v \ \chi) \qquad \forall \chi \in \Phi$$

$$wp.choose.true = true$$

$$wp.choose.\chi = (\exists v \ \chi) \qquad \forall \chi \in \Phi$$

This indicates that if there is a value assignment which satisfies χ, then *choose* will find it (behaves like *miracle*). Notice that *choose* is feasible, and is at one extreme of the feasible prescriptions

(under the ordering used by Back it is the greatest [BACR89]). The important thing about *choose* is that it is capable of finding a solution if one exists.

Both *miracle* and *choose* are not deterministic in the sense given in above. Instead they may be thought of as *hyper-deterministic* since they are overdetermined, meaning that they are able to perform the computations of several programs in one execution (*miracle* finds every result to every problem, and *choose* just those which exist). The following two programs are deterministic.

The first is *skip* [DIJE90] which is defined as

$$wlp.skip.\chi = \chi \qquad \forall \chi \in \Phi$$

$$wp.skip.true = true$$

$$wp.skip.\chi = \chi \qquad \forall \chi \in \Phi$$

Skip is the program which does not change state. It is like the no-op of an assembly language, and is the identity for sequential composition. Also *skip* is well-behaved and terminates on all inputs.

The other deterministic program discussed, on the other hand, never terminates. This program is called *abort* and is defined by

$$wlp.abort.\chi = true \qquad \forall \chi \in \Phi$$

$$wp.abort.true = false$$

$$wp.abort.\chi = false \qquad \forall \chi \in \Phi$$

That *abort* never terminates is indicated by the fact that $wp.abort.true = false$. This means that no executions from any initial state terminate. So *abort* does not terminate under any circumstance [DIJE90].

The dual of *choose* is *havoc* [DIJE90] (also called *fail* [BACR89], or *chance* [MORJ89]) and is defined by

$$wlp.havoc.\chi = (\forall v \ \chi) \qquad \forall \chi \in \Phi$$

$$wp.havoc.true = true$$

$$wp.havoc.\chi = (\forall v \ \chi) \qquad \forall \chi \in \Phi$$

The definition of *havoc* indicates that it will satisfy χ if all value assignments satisfy χ. So *havoc* is capable of finding an outcome if all outcomes are solutions.

The rest of the programs are constructed from a collection of deterministic programs. A quite powerful form of deterministic program, and the form used here, are assignment statements. Their meanings are defined by

$$wp.\text{“}y := E\text{”}.\chi = (y := E).\chi \qquad \forall \chi \in \Phi$$

$$wp.\text{“}y := E\text{”}.true = true$$

$$wlp.\text{“}y := E\text{”}.\chi = (y := E).\chi \qquad \forall \chi \in \Phi \qquad \text{[DIJE90]}.$$

The operators used in this construction are sequential composition, non-deterministic choice and its dual.

Sequential composition is denoted by a semi-colon and is defined on two programs p and q:

$$wlp.(p;q).\chi = wlp.p.(wlp.q.\chi) \qquad \forall \chi \in \Phi$$

$$wp.(p;q).true = wp.p.(wp.q.true)$$

$$wp.(p;q).\chi = wp.p.(wp.q.\chi) \qquad \forall \chi \in \Phi \qquad \text{[DIJE90]}.$$

If p and q are deterministic, so is their composition.

Non-deterministic choice is defined by

$$wlp.(p \sqcup q).\chi = wlp.p.\chi \wedge wlp.q.\chi \qquad \forall \chi \in \Phi$$

$$wp.(p \sqcup q).true = wp.p.true \wedge wp.q.true$$

$$wp.(p \sqcup q).\chi = wp.p.\chi \wedge wp.q.\chi \qquad \forall \chi \in \Phi \qquad \text{[DIJE90]}.$$

This definition may seem odd considering that choice is stated as “p or q.” But consider that both p and q must be able to achieve χ so both their pre-conditions must hold.

The dual of non-deterministic choice is called “demonic non-determinism” by Back [BACR89], because it finds the common behavior of the two programs (operands). It is defined as

$$wlp.(p \sqcap q).\chi = wlp.p.\chi \vee wlp.q.\chi \qquad \forall \chi \in \Phi$$

$$wp.(p \sqcap q).true = wp.p.true \vee wp.q.true$$

$$wp.(p \sqcap q).\chi = wp.p.\chi \vee wp.q.\chi \qquad \forall \chi \in \Phi$$

3.2.3. Orders. The elements of a specification language can be ordered based on the axiomatic semantics. There are three possible orders based on the weakest and weakest liberal pre-conditions. The first is based on *wp*, the second on *wlp* and the third on their combination. Each of these orderings could serve as an abstraction order to define the semantics of abstraction. The first is explained by both Back [BACR89] and Morris [MORJ89], and the latter is is developed in a similar fashion in Chapter 4.

Back and Morris both define the semantics of abstraction based upon the notion of state, which also supports predicate-based specification. Given a countable set of variables *Var* and a flat domain D of values (multiple types are ignored), a *state* is a mapping from *Var* to D which assigns each variable a value. The set of states (state space), ordered by the "equality relation" ($s_1 \leq s_2$ *iff* $s_1 = s_2$), is denoted *Sta* [MORJ89].

Another basic notion is the trivial Boolean lattice $B = \{0, 1\}$, for which $0 \leq 1$ (*false* is less than *true*). A *predicate* (or assertion) is an element of $[Sta \rightarrow B]$ (This is the same as $Sta \rightarrow B$ since states can only be equal), which is called *Pred*. *Pred* is a complete Boolean lattice under the order

$$\text{For } \phi, \psi \in Pred, \quad \phi \leq \psi \qquad \textit{iff} \qquad \text{for all states } \sigma, \quad \phi(\sigma) \leq \psi(\sigma)$$

(Notice that the elements of *Pred* are not all of the predicates in the sense of logic, rather each represents an equivalence class of such predicates, each of which is true for the same value assignments.) That *Pred* is a complete Boolean lattice comes from the fact that it's ordering is directly derived from that of B which is a complete Boolean lattice.

From *Pred*, the set of predicate transformers can be defined as $[Pred \rightarrow Pred]$. This set is denoted *Spec*, as is done by Morris [MORJ89], because it represents the set of predicate transformers corresponding to specification in the language discussed earlier. The predicate transformers are ordered by the extension of the order on *Pred*:

$$\text{For } s, t \in Spec, \qquad s \sqsubseteq t \quad \textit{iff} \quad s(\phi) \leq t(\phi) \qquad \text{for all predicates } \phi \text{ [BACR89].}$$

If s is a specification $\phi || \psi$, then $\phi || \psi \sqsubseteq t$ means that $\phi \rightarrow wp.t.\psi$. This is sufficient to argue that t "satisfies" the specification $\phi || \psi$, and that $\sqsubseteq$ is the desired abstraction order. Under this ordering *abort* is bottom and *miracle* is top.

The lack of distinction made between syntax and semantics can be confusing. For instance, Morris states and "proves" a theorem that "every specification" is an element of $\lceil Pred \rightarrow Pred \rceil$, thus embedding the specifications in *Spec* [MORJ89]. Unfortunately, this does not account for the fact that many specifications can possess the same predicate transformer semantics, and the proof is only for the predicate transformers. What can be embedded in *Spec* are the equivalence classes of specifications under a pre-order "made" by loosening the order on *Spec*.

Nelson [NELG89] defines a slightly different ordering than that of Morris, Morgan and Back just presented. He denotes the ordering derived here $\sqsubseteq_{wp}$, due to the close relationship with weakest preconditions and combines it with the ordering $\sqsubseteq_{wlp}$ defined using weakest liberal preconditions. These orders are defined for any statements p, q to be [NELG89]

$$p \sqsubseteq_{wp} q : wp.p.\phi \rightarrow wp.q.\phi \text{ for any } \phi.$$

$$p \sqsubseteq_{wlp} q : wlp.p.\phi \rightarrow wlp.q.\phi \text{ for any } \phi.$$

$$p \sqsubseteq q : p \sqsubseteq_{wp} q \text{ and } p \sqsubseteq_{wlp} q.$$

3.3. Syntactic Foundations.

The syntactic aspect of the ARM is derived from the Laws of Programming which are syntactic laws of non-determinism [HOAC87]. The specification language used in both is a non-deterministic language extended by the inclusion of the dual of the non-determinism operator. This extension introduces infeasible programs to the language. Also desirable is that the language be strict and interactive since both are properties of languages used in practice. These properties influence the Laws of Programming in ways that are not as important to the ARM, and are only briefly mentioned.

The extended language is denoted $\mathcal{L}$, and has three sub-languages which are of interest. These are the deterministic subset $\mathcal{L}_{Det}$, the non-deterministic language $\mathcal{L}_{Non-Det}$, and the language of "infeasible" programs $\mathcal{L}_{Inf}$. The language $\mathcal{L}_{Det}$ is also a sub-language of both $\mathcal{L}_{Non-Det}$ and $\mathcal{L}_{Inf}$, because these languages are the closure of $\mathcal{L}_{Det}$ on $\sqcup$ and $\sqcap$ respectively.

In the following the properties of $\mathcal{L}$ are described using the Laws of Programming [HOAC85] [HOAC87] [HOAC89]. These laws also apply to the sub-languages as appropriate.

3.3.1. Abstraction Ordering. One of the most important features of the Laws of Programming is the "abstraction" order. This ordering is defined (opposite that of [HOAC89] but consistent with [HOAC85] and [HOAC87]) as follows:

> If p and q are programs, then $p \sqsubseteq q$ means p may be substituted for q and will perform as well as, or better, than q [HOAC87].

This definition is intentionally ambiguous on Hoare's part since the ordering means different things for different languages. In particular, he states that the interpretation of $\sqsubseteq$ differs between deterministic and non-deterministic languages. For the ARM it is necessary that the order's properties for deterministic languages are consistent with those for the non-deterministic language (perhaps a "subset"). The precise meaning of the order is sufficiently addressed by the intended semantics, but is loosely interpreted here to represent "is a realization of."

The order has the properties of reflexivity (any specification is its own realization) and transitivity (the realization of a realization of p is also a realization of p). Anti-symmetry does not hold for the abstraction order (so it is a pre-order) because programs can be semantically equivalent but syntactically non-identical. This equivalence is formed when two specifications are mutual realizations (like for the intended semantics), and is denoted by $\equiv$.

Under this order, there are two distinguished specifications. These are the top, $\top$, and bottom, $\bot$ (and correspond to those described in the informal intended semantics). The top program is one such that $\top \in \mathcal{L}$, and $\forall p \in \mathcal{L}$, $p \sqsubseteq \top$. This is the most non-deterministic program, and could do anything that any program in $\mathcal{L}$ could do. Hoare says this is "the most useless of all programs," [HOAC89] which is true because it cannot be relied upon to ever do any particular one of its tasks.

The bottom program, on the other hand, is the most "deterministic" (this is *miracle* so it is "hyper-deterministic"). This program is the realization of all other programs: $\forall p \in \mathcal{L}$, $\bot \sqsubseteq p$. So in theory, $\bot$ is the ultimate program that solves all problems. Hoare argues that the inclusion of such a program in a language makes the language "futile" [HOAC89]. While the ontology of the bottom is not clear, it does seem that we should be able to take the meet of any number of programs. Rather paradoxically, $\bot$ is interpreted as being defective in some way, i.e. short circuited. Another perspective is that $\bot$ is impossible to construct; thus even if it is "part" of the language it is not expressible.

3.3.2. Join and Meet. Assuming the membership of $\top$ and $\bot$ in $\mathcal{L}$, the meet and join operators can be defined. The join is the non-deterministic choice operator, but the meet is not usually included in a language. Because the language has been extended to serve as a specification language the meet is included in its syntax.

The meet of p and q is their ***greatest common realization*** and is denoted $p \sqcap q$. The Law which defines the meet is

$$\forall p, q, r \qquad r \sqsubseteq p \sqcap q \quad \textit{iff} \quad r \sqsubseteq p \ \& \ r \sqsubseteq q$$

Equivalently, the meet can be defined for programs p and q by

$$p \sqcap q \equiv q \quad \textit{iff} \quad q \sqsubseteq p$$

Dual to the meet is the join operator (i.e. "choice") and it is defined similarly:

$$\forall p, q, r \qquad p \sqcup q \sqsubseteq r \quad \textit{iff} \quad p \sqsubseteq r \ \& \ q \sqsubseteq r$$

and for programs p and q

$$p \sqcup q \equiv q \quad \textit{iff} \quad p \sqsubseteq q$$

The meet and join are also idempotent, commutative, and associative:

$$p \sqcup p \equiv p \qquad p \sqcap p \equiv p$$

$$p \sqcup q \equiv q \sqcup p \qquad p \sqcap q \equiv q \sqcap p$$

$$(p \sqcup q) \sqcup r \equiv p \sqcup (q \sqcup r) \qquad (p \sqcap q) \sqcap r \equiv p \sqcap (q \sqcap r)$$

Additionally, top is the unit and bottom the zero as shown by:

$$p \sqcup \top \equiv \top \qquad p \sqcup \bot \equiv p$$

$$p \sqcap \top \equiv p \qquad p \sqcap \bot \equiv \bot$$

3.3.3. Programing Operators. The other operator that is discussed is sequential composition. This is the familiar semi-colon from imperative languages such as Pascal or Ada. The operator allows the definition of a sequence of statements. This composition operator is the primary means of constructing larger programs presented here. This and the other means of composing programs, including alternation and loop constructors, are not crucial to the definition of the structure of the

ARM. What is important, however, is that these operators preserve the level of abstraction of their operands. The property of monotonicity ensures this.

Sequential composition gives an indication of the properties that these operators have. The properties of sequential composition are associativity, existence of an identity, and monotonicity.

(1) $p;(q;r) = (p;q);r$

(2) $\exists i\ \forall p\ p;i = p = i;p$

(3) $\forall p,q,r$ if $p \sqsubseteq q$ then $p;r \sqsubseteq q;r$ and $r;p \sqsubseteq r;q$

The last property is monotonicity and is part of a requirement that Hoare makes, that all operators of the language must be monotonic with respect to the abstraction order [HOAC89]. Without this requirement, the abstraction order would not be useful in modeling the realization of specifications.

The identity of sequential composition is *skip* since it does not affect the values of any variables. Top is the zero of composition in a non-interactive language, but is only a "pseudo-zero" in an interactive language. This is because in an interactive program $(q;\top)$, q can postpone the "fate" of $\top$ indefinitely.

A useful property of sequential composition is distribution across the join and meet.

$$\forall p,q,r,s \qquad r;(p \sqcup q);s \equiv (r;p;s) \sqcup (r;q;s)$$

The statement of this law indicates that composition distributes both ways.

3.4. Foundational Synthesis.

The material in this chapter combines to form the foundation of the ARM. This section summarizes these observations and assumptions relevant to the ARM. Chapter 4 then builds on these foundations in the construction of the ARM.

3.4.1. Observations. While the foundational material covers both syntax and semantics, the most important observations regard the semantics. These concern the semantics of abstraction.

The first observation is about the order of the prescriptions *miracle*, *choice*, *skip*, *abort*, and *havoc* defined above. Three orders based on weakest pre-condition and weakest liberal pre-conditions were defined as well. Under each ordering the prescriptions form different sequences of abstractions.

The ordering, $\sqsubseteq_{wp}$, based purely on weakest pre-conditions is that defined by Back [BACR89] and Morris [MORJ89]. It structures the prescriptions as

$$abort \sqsubseteq_{wp} chance \sqsubseteq_{wp} skip \sqsubseteq_{wp} choose \sqsubseteq_{wp} miracle$$

Note that under this ordering, *abort*, the non-terminating program is the bottom. The ordering based on weakest liberal preconditions, $\sqsubseteq_{wlp}$, orders them as

$$chance \sqsubseteq_{wlp} skip \sqsubseteq_{wlp} choose \sqsubseteq_{wlp} miracle \equiv_{wlp} abort$$

Under this ordering *abort* is the top, and the rest are the same. The ordering defined by combining these orders, suggested by Nelson [NELG89], therefore has a similar structure.

$$chance \sqsubseteq skip \sqsubseteq choose \sqsubseteq miracle$$

The main difference is that abort relates only to *miracle*:

$$abort \sqsubseteq miracle.$$

The problem with this ordering is that it has no bottom and so is not complete, posing a problem in Chapter 4 where the semantics of abstraction is proved to be a complete lattice. Another prescription is required to complete the order.

Consider a non-deterministic language with assignment statements forming the deterministic base language. Let the most non-deterministic program be called *bot*. If *bot* is the most non-deterministic program, an equivalent program exists which is the "choice" of deterministic programs. The predicate transformer semantics of *bot* can then be constructed from the deterministic programs.

The "choice" of *skip* and *abort* is a good start since both programs are deterministic. The definition of this program is:

$$wlp.(skip \sqcup abort).\chi = \chi \wedge true = \chi \qquad \forall \chi \in \Phi$$

$$wp.(skip \sqcup abort).true = true \wedge false = false$$

$$wp.(skip \sqcup abort).\chi = \chi \wedge false \qquad \forall \chi \in \Phi$$

With this program in hand, all that is required to find *bot* is to discover a combination of assignments such that the *wlp* of their choice is *false*.

To do this, the nature of the first-order language L must be expanded. In order for this language to be useful for reasoning about programs, the terms must relate the variables to values from the domain. Therefore the formulae of the language are predicates which are satisfied by a set of values. For each of these predicates, with the exception of ***true***, at least one value assignment does not satisfy the predicate.

This provides a means of asserting the existence of a choice of assignment statements whose *wlp* is *false*. Simply note that the weakest pre-condition of a non-deterministic choice statement is the conjunction of the weakest pre-conditions of its component parts. The desired program is formed by the choice of programs such that for each equivalence class of predicates, the program is an assignment which does not satisfy the predicates in the class. The final result is a program for which *wlp* is *false* for every predicate. This combined with the program formed from ***skip*** and *abort* is *bot* and is defined as

$$wlp.bot.\chi = false \qquad \forall \chi \in \Phi$$

$$wp.bot.true = false$$

$$wp.bot.\chi = false \qquad \forall \chi \in \Phi$$

This definition means that there are no initial states of computations that either terminate or don't terminate. Alternatively, there are no computations of *bot* that guarantee a final condition.

So *bot* can be considered the most non-deterministic program (albeit infinite). This argument suggests that the weakest pre-conditions of *bot* are *false* because no final outcome can be guaranteed from any given input. *Bot* behaves unpredictably, as the most non-deterministic program should.

The addition of *bot* completes the orders. First note that *bot* $\equiv_{wp}$ ***abort*** and that *bot* $\sqsubseteq_{wlp}$ ***chance***. Together these help to show that Nelson's order has *bot* as bottom, and that ***abort*** is also related to *bot*:

$$bot \sqsubseteq chance \sqsubseteq skip \sqsubseteq choose \sqsubseteq miracle$$

$$bot \sqsubseteq abort \sqsubseteq miracle$$

Clearly, the distinction between the three orderings is the handling of non-termination (***abort***). Which version is appropriate is decided in Chapter 4. An observation related to this decision is made by considering the conjugates of the prescriptions and the nature of non-determinism.

Recall that a program p is deterministic if $wp.p$ and $wlp.p$ are conjugates. Of the prescriptions considered, ***abort*** and ***skip*** are deterministic, and so are self-dual. The others have duals, although they are not self-dual.

The dual of *bot* is *miracle*, and the dual of *chance* is *choose*. These are easily checked by finding the conjugate of the definition of each. Existence of the duals suggests a structure based on determinism with "hypo-deterministic" programs at the top, deterministic programs in the middle, and "hyper-deterministic" programs at the bottom. (hypo- meaning non-deterministic, and hyper-deterministic meaning infeasible.) The order suggested by Nelson matches this structure the best.

3.4.2. Assumptions. The Abstraction Refinement Model is strongly based upon language, so most assumptions regard language issues. The background for these assumptions is discussed in this and the last chapter. The two essential assumptions are (1) a useful relationship exists between specification and non-determinism, and (2) a wide-spectrum language can be used without loss of generality.

The language used does not include any iterative or recursive constructs. The primary reason is that iteration and recursion are typically defined as limits on programs with respect to the abstraction order. These constructs are essentially extensions of the notation to abstract away from lengthy straight-line programs, much like logical quantifiers abstract away from lengthy disjunctions and conjunctions. Eventually, the inclusion of these constructs may be desirable, but nothing is lost by their exclusion.

3.4.2.1. Specification and Non-determinism. A natural correspondence between the abstraction and non-determinism is suggested in Chapter 2 by the phrase "ambiguity as choice." It seems clear the ambiguity of a specification provides the opportunity for choice of detail. So this ambiguity might be made explicit by a choice of the allowed details. The assumption based on this reasoning is that a specification language can be replaced by an equivalent non-deterministic language. (This is also a basic assumption of the Laws of Programming [HOAC87].)

This assumption is also justified through semantics, although the connections between the semantics of "abstraction" and non-determinism are not well-established. For example, the correspondence between axiomatic semantics and denotational semantics is only made explicit at the program level [PLOG80] [APTK86]. A higher-level correspondence is only implicit in the work of Nelson [NELG89]. Both of these are limited to axiomatic and denotational semantics (power domains in particular), ignoring other models of non-determinism.

The clearest relationship is between the relational model and the axiomatic semantics. Each utilizes the notion of state. In the relational model, the states are an explicit part of the formalism since they make up the relation. In the axiomatic model, the states are allowed by the pre- and post-conditions. So the use of states implies the correspondence between the models.

3.4.2.2. Wide-spectrum vs. Narrow-Spectrum. Another language issue raised in Chapter 2 is the distinction between wide- and narrow-spectrum languages. The language axiomatized by the Laws of Programming, and used here, is a wide-spectrum language. Although in practice a collection of narrow-spectrum languages is more likely to be used, the inhomogeneity causes difficulties. The homogeneity of a wide-spectrum language makes the connections between levels of abstraction easier to understand and relate to, making it more appropriate for this initial study.

The assumption is that a wide-spectrum language is sufficient to express all possible system descriptions. This does seem to be the case, since an assertional language with prescriptions is the most general of its kind (all others being a sub-language).

Chapter 4. Model Development

With the foundation complete, the model can be described and developed. The Abstraction Refinement Model as originally defined [NANR89] shows strong influences of both the Draco paradigm and Laws of Programming. The nature of the model remains the same although the details have been refined and formalized.

The Abstraction Refinement Model connects the ideas from Draco and the Laws of Programming rather naturally. Within the Draco paradigm the *domain structure graph* (a directed acyclic graph) is used to represent the possible "states" of the product, and the design decisions (or transformations) between them. In the Laws of Programming the abstraction ordering gives the language its structure. This "lattice" structure can be drawn as a Haase diagram which is a directed acyclic graph, where an arc exists from a node p to a node q if $p \leq q$ (direction of arcs is downward; no arrow is used). (Note that no properties of this structure have been established and that a lattice requires that both join and meet be well defined.) This suggests that the Laws of Programming describe a "domain structure graph" based upon the non-deterministic language. In Chapter 5 it is assumed that such a structure can be imposed on arbitrary descriptive formalisms.

In this more restricted graph, design decisions (transformations) correspond to the $\leq$-edges in the Haase diagram. Such a transformation performs refinement of the descriptive object it acts upon: it moves the description from an abstract form to a more concrete form (thus imparting the name *Abstraction Refinement Model*). This combination of Draco and the Laws of Programming combine two very important notions and forms the basis of the Abstraction Refinement Model (ARM).

Despite the influence that Draco and the Laws of Programming have on the ARM, these ideas appear in other sources in the literature as well. This observation is certainly justified by Chapter 2. The extent to which the underlying concepts of the ARM can be drawn from the literature is encouraging.

This chapter develops the ARM, paying special attention to the details of its definition. The points considered include the intended semantics (i.e. semantics of abstraction, or realizability models), the syntactic aspects, and the correspondence between these. The intended semantics provide the intuition and justification for the structure of the ARM developed in the syntax. Correspondence between these structures provides an interpretation to the ARM.

4.1. Intended Semantics.

The ARM is a syntactic model based on laws formalizing semantic equivalence of syntactic structures (the Laws of Programming [HOAC85] [HOAC87] [HOAC89]). Despite the relation to semantics, the ARM does not depend on any particular semantic model. The intent of this section is to introduce a semantic interpretation which justifies the structure of the ARM (i.e. to prove there exists a semantics with the right structure that is consistent with the desired interpretation).

This semantic model is meant primarily to give a precise meaning to the abstraction order on which the ARM relies. The semantic model is called the "intended" semantics merely because it represents the informal interpretation of the model. The discussion first gives the informal and then the formal semantics of the ARM. Later an isomorphism between the two models is developed.

4.1.1. Informal Discussion. The ARM is a syntactic model of the software evolution process. However, at the most fundamental level it is simply an algebra of programs. This algebra could represent a number of different things, but to form the ARM a meaning is attributed to the structure.

The algebra on which the ARM is based is a complete distributive lattice. By letting the objects of the algebra be (equivalence classes of) system descriptions, the basic structure of the ARM is formed. The order of the lattice is the abstraction order which indicates that one system description is an implementation of another. So the operations of meet and join find common realizations and specifications respectively. The lattice structure of the class of system descriptions (a specification language) can be explained intuitively.

Consider any two system descriptions a and b; they have a common abstraction which is a "choice" between them. This abstraction says roughly "either do a or do b." A common realization of the same two system descriptions would mean "behave like both a and b." These are the join and meet necessary to form a lattice.

Completeness of the lattice requires a top and bottom. It is possible to conceive of a system description which is the most abstract or most general. This simply specifies that "something" should be done, not what the something is or how to do it. In terms of non-determinism, this would be the most non-deterministic program, because any eventuality (possible in the language) could occur (this program would be the non-deterministic choice of all other programs). Thus every system description is a realization of this *top* system description. The least system description, or the *bottom*, is also the realization of every other system description, and it corresponds exactly to Dijkstra's *miracle*. The magic of the bottom is lost, however, when it is observed that it is the meet

of all system descriptions and has their common behavior. But the only behavior in common with all programs is none. (This corresponds to the role of the empty set in a Boolean Set Algebra).

The distributivity of the lattice of system descriptions is not easily explained intuitively. Note also that the existence of the meet for all systems descriptions is not clear because of its relationship to infeasibility. To understand the latter the structure must be examined more closely.

Beneath the top there are system descriptions which are more concrete (more deterministic). These in turn, have concrete (deterministic) realizations which satisfy them. This layer of concrete realizations is the level which software development tries to achieve. At the concrete level the system descriptions are all possible "executable" programs, including those which behave badly (e.g. don't terminate).

All the system descriptions from the top to the concrete level are "feasible" (vaguely meaning implementable). However, not all specifications are feasible; especially those expressed in a "natural" language. These infeasible system descriptions complete the lower half of the lattice.

An infeasible system description is partial in the sense that it specifies no behavior for some inputs. The lack of behavior is what makes it infeasible, since no operational semantics can be given for it. These system descriptions are the meets of feasible system descriptions, so the meet can be interpreted as finding a system description which has the behavior common to its operands (an "intersection" of behavior).

In most respects the lattice corresponds to intuition about how system descriptions should be ordered by the abstraction order. One difficulty is that the infeasible system descriptions comprise the lower portion of the lattice. It is not obvious that all infeasible programs are realizations of feasible programs. Unfortunately, this problem cannot be easily resolved without a study of feasibility that is not possible here. One solution is to just throw the infeasible system descriptions out and work with a semi-lattice. However, for the purposes of this research the validity of the notion of infeasibility is not critical, for the sake of completeness faith is placed in Hoare [HOAC87] and the infeasible system descriptions are assumed to be realizations of feasible ones.

Beyond infeasibility, there is the question of *degeneracy* of system descriptions. One of the goals of the ARM is to be able to model both the good and bad instantiations of a system. The bad instantiations are captured in the notion of degeneracy.

Recall from Chapter 2 the concept of failure. A failure is an operational deviation from the behavior intended by the specification. The notion of degeneracy could be entirely founded on

failures. However, programs may exist with behavior which, although given by the specification, is perceived as a failure. In this case, the fault would be in the specification rather than the program since the specification deviates from the perceived view of how the system should behave (i.e. the customer is right).

Degeneracy, then can be of two types. *Non-conformance* is the property that the current specification is not a realization of of its predecessor, or is not *verifiable*. The more complex property that the initial specification is incorrect but the realizing specifications are correct is *invalidity*. Therefore, the notions of degeneracy are related to those of verification and validation.

These notions of degeneracy can be symbolically represented with the abstraction order $\sqsubseteq$. Let p be a program and q its specification, with u as the user's view of the system. Then p is invalid if $p \not\sqsubseteq q$, and non-conforming if $p \sqsubseteq q$ but $q \not\sqsubseteq u$. That degeneracy can be represented in this way suggests that the abstraction order can sufficiently describe "badness" of a system description and that no further effort is required.

The following is expected of a formal statement of the intended semantics:

(1) A precise statement of the meaning of $p \sqsubseteq q$

(2) The model should have the structure of a complete, distributive lattice based on $\sqsubseteq$.

(3) The top and bottom have a consistent interpretation with that given above.

Such a model is developed in the next section.

4.1.2. Formal Definition. The formal development of the intended semantics of the ARM is based on predicate transformers and is much like that of Back and Morris discussed in Chapter 3. The discussion of the informal semantics suggests that the ordering proposed by Nelson be used, and there is enough difference to merit a second derivation.

This treatment is based on the recognition that many different formulas of a first-order (logical) language can be equivalent. So the basis of the model is the Lindenbaum algebra of a first-order language L. The Lindenbaum algebra is the Boolean algebra of equivalence classes of formulas [BELJ77] as described in Chapter 3.

The semantic model is constructed by showing the set of monotonic predicate transformers is a complete distributive lattice, and then this structure is transferred to the weakest precondition semantics of the language from Chapter 3.

4.1.2.1. Algebra of Predicate Transformers. The algebra of predicate transformers is con-

structured from the set of monotonic predicate transformers and the Lindenbaum algebra on a first-order language L. This algebra has as elements equivalence classes of predicate transformers, which can be interpreted as functions on the Lindenbaum algebra of L. The goal is to define the structure of the intended semantics of the ARM.

The abstraction order on predicate transformers is defined by

$$f \sqsubseteq_p g \quad \textit{iff} \quad \vdash f(\phi) \rightarrow g(\phi) \qquad \forall \phi \in \Phi$$

This order is a pre-order (reflexive and transitive) since implication has these properties. An equivalence relation on predicate transformers can be defined from this order

$$f \equiv_p g \quad \textit{iff} \quad f \sqsubseteq_p g \;\&\; g \sqsubseteq_p f$$

or equivalently

$$f \equiv_p g \quad \textit{iff} \quad f(\phi) \approx g(\phi) \qquad \forall \phi \in \Phi$$

The relation $\equiv_p$ is an equivalence relation, following from the fact that $\approx$ is the equivalence of the Lindenbaum algebra.

The two operators $\sqcap$ and $\sqcup$ on predicate transformers act like join and meet under the equivalence. They are defined for all $\phi \in \Phi$ by

$$(f \sqcup g)(\phi) = f(\phi) \vee g(\phi)$$

$$(f \sqcap g)(\phi) = f(\phi) \wedge g(\phi)$$

The following theorem shows that these operators act like join and meet.

THEOREM 4.1.

(1) $f \sqcup g \equiv_p g \quad \textit{iff} \quad f \sqsubseteq_p g$

(2) $f \sqcap g \equiv_p g \quad \textit{iff} \quad g \sqsubseteq_p f$

PROOF:

Only the first is shown, appealing to the principle of duality for the second. Let $f \sqcup g \equiv_p g$ (show $f \sqsubseteq_p g$), then $(f \sqcup g)(\phi) \approx g(\phi)$, for all $\phi \in \Phi$. Which implies

$$|f(\phi)| \vee |g(\phi)| = |g(\phi)| \qquad \forall \phi \in \Phi$$

but since $B(L)$ is a Boolean algebra and $\vee$ the join

$$|f(\phi)| \leq |g(\phi)| \qquad \forall \phi \in \Phi$$

or equivalently $\vdash f(\phi) \rightarrow g(\phi)$, $\forall \phi \in \Phi$, which means $f \sqsubseteq_p g$. Q.E.D.

The equivalence classes of predicate transformer f are represented as

$$[f] = \{g : \Phi \rightarrow \Phi \mid g \text{ monotonic, and } g \equiv_p f\}$$

These classes may be considered mappings from $B(L)$ to $B(L)$ with the following definition

$$[f](|\phi|) = |f(\phi)|$$

Note that if the definition $[f](|\phi|) = \{g(\phi) \mid g \equiv f \ \& \ \psi \approx \phi\}$ were used instead, the mapping would be to a subset of $|f(\phi)|$: $[f](|\phi|) \subseteq |f(\phi)|$. This is because for $g(\phi)$ to be in $|f(\phi)|$ it is not necessary that $g \equiv_p f$. Either way, the equivalence classes can be considered monotonic mappings on $B(L)$. To see this let $|\psi| \leq |\phi|$, and show $[f](|\psi|) \leq [f](|\phi|)$. The assumption $|\psi| \leq |\phi|$ means $\vdash \psi \rightarrow \phi$, and f monotonic means $\vdash f(\psi) \rightarrow f(\phi)$. Therefore, $|f(\psi)| \leq |f(\phi)|$, which by definition is equivalent to $[f](|\psi|) \leq [f](|\phi|)$.

These "mappings" on the Lindenbaum algebra are used to build the "semantic entities" intended to correspond to the equivalence classes of programs in the ARM. The partial order on the equivalence classes of predicate transformers is defined from the pre-order $\sqsubseteq_p$:

$$[f] \sqsubseteq [g] \quad \textit{iff} \quad f \sqsubseteq_p g.$$

Given this ordering, the objective is to show that the set of equivalence classes is a lattice. This is done by showing that the "join" and "meet" on the predicate transformers induces the join and meet on the equivalence classes of predicate transformers.

LEMMA 4.2. *The join on the equivalence classes* $[f]$ *and* $[g]$ *is* $[f \sqcup g] = \sup\{[f], [g]\}$, *and the meet* $[f \sqcap g] = \inf\{[f], [g]\}$.

PROOF: Once again, only the proof for $\sqcup$ is given, with the proof for $\sqcap$ an appeal to duality.

Because $\sqcup$ acts like the join (see the preceding theorem), $f \sqsubseteq_p f \sqcup g$ and $g \sqsubseteq_p f \sqcup g$, which together imply $[f] \sqsubseteq [f \sqcup g]$ and $[g] \sqsubseteq [f \sqcup g]$. So $[f \sqcup g]$ is an upper bound of $[f]$ and $[g]$; that $[f \sqcup g]$ is the least upper bound is required.

Let $[h]$ be a least upper bound of $[f]$ and $[g]$. Then $|h(\phi)|$ is a least upper bound of $|f(\phi)|$ and $|g(\phi)|$, which means $|h(\phi)| = |f(\phi)| \vee |g(\phi)|$ for all $\phi \in \Phi$. But

$$\begin{aligned}[f \sqcup g](|\phi|) &= |f(\phi) \vee g(\phi)| \\ &= |f(\phi)| \vee |g(\phi)| \\ &= |h(\phi)|\end{aligned}$$

By the equivalence of formulas

$$(f \sqcup g)(\phi) \approx h(\phi) \quad \forall \phi \in \Phi$$

which by definition means $[f \sqcup g] = [h]$. So $[f \sqcup g] = \sup\{[f], [g]\}$. Q.E.D.

If $\sup\{[f], [g]\}$ is denoted by $[f] \sqcup [g]$ and $\inf\{[f], [g]\}$ by $[f] \sqcap [g]$, then

$$\begin{aligned}[f \sqcup g] &= [f] \sqcup [g] \\ [f \sqcap g] &= [f] \sqcap [g]\end{aligned}$$

With these results the set $P(L) = \{[f] \mid f : \Phi \rightarrow \Phi, \text{ monotonic}\}$ is a lattice with join $\sqcup$ and meet $\sqcap$. The construction of the algebra of predicate transformers is essentially finished. All that remains is to show that $P(L)$ is complete and distributive.

LEMMA 4.3. $\langle P(L), \sqcap, \sqcup \rangle$ *is distributive.*

PROOF: The objective is to show that

$$[f] \sqcap ([g] \sqcup [h]) = ([f] \sqcap [g]) \sqcup ([f] \sqcap [h])$$

Once this is established an appeal to the duality principle gives the dual.

Because these are equivalence classes, it must be shown that all elements of each side belong to the other. This can be done for both sides simultaneously. Let $l \in [f] \sqcap ([g] \sqcup [h])$ which by definition is equivalent to

$$\begin{aligned} |l(\phi)| &= ([f] \sqcap ([g] \sqcup [h]))(|\phi|) \qquad \forall \phi \in \Phi \\ &= |f(\phi)| \wedge ([g] \sqcup [h])(|\phi|) \qquad \forall \phi \in \Phi \\ &= |f(\phi) \wedge (g(\phi) \vee h(\phi))| \qquad \forall \phi \in \Phi \end{aligned}$$

which by distributivity of $\wedge$ and $\vee$, equals $|(f(\phi) \wedge g(\phi)) \vee (f(\phi) \wedge h(\phi))|$. By definition of the equivalence of predicate transformers this means $l \in ([f] \sqcap [g]) \sqcup ([f] \sqcap [h])$.

Therefore, $l \in [f] \sqcap ([g] \sqcup [h])$ whenever $l \in ([f] \sqcap [g]) \sqcup ([f] \sqcap [h])$. The dual holds by the duality principle. Q.E.D.

THEOREM 4.4. *$\langle P(L), \sqcap, \sqcup \rangle$ is complete.*

PROOF: The proof must show a top and bottom exist. This is shown using the following claims:

(1) the top is $\top = [f]$ such that $|f(\phi)| = 1 \quad \forall \phi \in \Phi$

(2) the bottom is $\bot = [f]$ such that $|f(\phi)| = 0 \quad \forall \phi \in \Phi$

Prove the claim for top. Assume $[g] \sqsupseteq [f] = \top$. Then $|g(\phi)| \geq |f(\phi)|$ for all $\phi \in \Phi$. But $|f(\phi)| = 1$ which is the top of $B(L)$, so $|g(\phi)| = 1$ as well. This means $|f(\phi)| = |g(\phi)|$ for all $\phi \in \Phi$, and therefore $[g] = [f]$. Dually, for bottom assume there is a class less than it. Since bottom corresponds to the bottom of $B(L)$ no equivalence class can be lower in $P(L)$.

It must also be shown that all equivalence classes are bounded by $\top$ and $\bot$. For top observe that for an arbitrary monotonic predicate transformer f and $g \in \top$, that $|f(\phi)| \leq |g(\phi)|$, for all $\phi \in \Phi$. So $[f] \sqsubseteq [g] = \top$ for any f. Similarly, for bottom, it is the case that for arbitrary f and $g \in \bot$ that $|g(\phi)| \leq |f(\phi)|$, for all $\phi \in \Phi$. So $\bot = [g] \sqsubseteq [f]$. Q.E.D.

The proof of this Lemma concludes the proof that $\langle P(L), \sqcap, \sqcup \rangle$ is a complete distributive lattice. This is the desired structure of the intended semantics. Note that if non-monotonic predicate transformers were allowed complements would be possible and this would be a Boolean algebra.

4.1.2.2. Ordering Programs. Given that the equivalence classes of monotonic predicate transformers are the elements of a complete, distributive lattice, the ordering of programs is now possible.

The constructors *wp* and *wlp* are in fact mappings from a language to the monotonic predicate transformers. The derivation of the order on programs using these mappings is much like the definition of the order on predicate transformers.

Notice that a structure of semantic "entities" is not constructed. Although this would have been much cleaner, it does not seem to be possible using axiomatic semantics. As suggested in Chapter 2, both $wp.p$ and $wlp.p$ are required to describe the total behavior of the program p. So the order is defined in terms of both, as was suggested by Nelson [NELG89]. The result is that the programs are ordered consistently with the intended semantics.

The approach is to define the order and consider the structure induced on the prescriptions of Chapter 3. This allows a comparison of the semantic model with the informal intended semantics.

4.1.2.2.1. Ordering. Earlier it is shown that the monotonic predicate transformers could be ordered in such a way that they form a complete distributive lattice. The constructors *wp* and *wlp* can be considered functions from the programs into the lattice of predicate transformers. These functions can be used to define orderings on programs by extending the order on predicate transformers, similar to the work of Back [BACR89] and Morris [MORJ89], but the dual corresponds better with the informal semantics. So the orderings are defined as

$$p \sqsubseteq_{wp} q \quad \textit{iff} \quad wp.p \sqsupseteq_p wp.q$$

$$p \sqsubseteq_{wlp} q \quad \textit{iff} \quad wlp.p \sqsupseteq_p wlp.q$$

Together *wp* and *wlp* define the total behavior of programs, so the desired ordering is the one suggested by Nelson.

$$p \sqsubseteq q \quad \textit{iff} \quad p \sqsubseteq_{wp} q \;\&\; p \sqsubseteq_{wlp} q$$

The order induces a complete, distributive lattice on equivalence classes of programs. This is ensured by the nature of *wp* and *wlp*, which are homomorphisms from a language into the monotonic predicate transformers. The homomorphism is given by the definitions of the language operators (page 38) and the join and meet on predicate transformers (page 52). Together these imply for programs p and q

$$wp.(p \sqcap q) \equiv_p wp.p \sqcup wp.q$$

$$wp.(p \sqcup q) \equiv_p wp.p \sqcap wp.q$$

which are stated similarly for wlp.

By defining equivalence in the usual way it can be seen that for programs p and q that

$$[p] = [q] \quad \textit{iff} \quad [wp.p] = [wp.q] \ \ \& \ \ [wlp.p] = [wlp.q]$$

These definitions allow a direct transfer of the properties of $P(L)$ to the set of equivalence classes of programs $\mathcal{L}/_\equiv$

For example consider distributivity:

$$[p] \wedge ([q] \vee [r]) = ([p] \wedge [q]) \vee ([p] \wedge [r])$$

where p, q and r are programs and $\vee$ and $\wedge$ are the join and meet on $\mathcal{L}/_\equiv$. Because $P(L)$ is distributive

$$[wp.p] \wedge ([wp.q] \vee [wp.r]) = ([wp.p] \wedge [wp.q]) \vee ([wp.p] \wedge [wp.r])$$

and

$$[wlp.p] \wedge ([wlp.q] \vee [wlp.r]) = ([wlp.p] \wedge [wlp.q]) \vee ([wlp.p] \wedge [wlp.r])$$

together these translate to the desired result (given well-definedness of $\vee$ and $\wedge$). In a similar fashion the other properties can be proved to show that $\mathcal{L}/_\equiv$ is also a complete, distributive lattice.

4.1.2.2.2. Structure. Additional structure can be placed on the language using the axiomatic semantics. In particular, it can be "leveled" in terms of "abstraction." The result given here shows that the deterministic sub-language forms such an abstraction level. Further levels can also be defined and the levels ordered, and this is indeed done for the syntactic model toward the end of the chapter.

First the following result about the conjugate and the order on predicate transformers is needed.

LEMMA 4.5. *For predicate transformers f and g, $f \sqsubseteq_p g \quad$ iff $\quad f^* \sqsupseteq_p g^*$*

PROOF:

$$\begin{aligned} f \sqsubseteq_p g &\Leftrightarrow f(\phi) \rightarrow g(\phi) \qquad \forall \phi \in \Phi \\ &\Leftrightarrow f(\neg\phi) \rightarrow g(\neg\phi) \qquad \forall \phi \in \Phi \\ &\Leftrightarrow \neg g(\neg\phi) \rightarrow \neg f(\neg\phi) \qquad \forall \phi \in \Phi \\ f^* \sqsupseteq_p g^* &\Leftrightarrow g^*(\phi) \rightarrow f^*(\phi) \qquad \forall \phi \in \Phi \text{Q.E.D.} \end{aligned}$$

Actually, only the underlying property for implications is used. The desired result tells something about the structure of the set of deterministic programs with respect to Nelson's order [NELG89].

THEOREM 4.6. *The deterministic programs are either incomparable or equivalent.*

PROOF: Let p and q be deterministic programs such that $p \sqsubseteq q$. Then $p \sqsubseteq_{wp} q$ and $p \sqsubseteq_{wlp} q$, meaning $wp.p.\chi \rightarrow wp.q.\chi$ for all $\chi \in \Phi$ and $wlp.p.\chi \rightarrow wlp.q.\chi$ for all $\chi \in \Phi$. Because p is deterministic, $wp.p$ and $wlp.p$ are conjugates, and similarly for q. So $(wp.p)^*$ can be substituted for $wlp.p$. Substituting the conjugates in both formulae results in $(wp.p)^*.\chi \rightarrow (wp.q)^*.\chi$ for all $\chi \in \Phi$ and $(wlp.p)^*.\chi \rightarrow (wlp.q)^*.\chi$ for all $\chi \in \Phi$. By the Lemma, this is equivalent to $wp.q.\chi \rightarrow wp.p.\chi$ for all $\chi \in \Phi$ and $wlp.q.\chi \rightarrow wlp.p.\chi$ for all $\chi \in \Phi$. Therefore, $q \sqsubseteq_{wp} p$ and $q \sqsubseteq_{wlp} p$, result in $q \sqsubseteq p$. But since $p \sqsubseteq q$, $p \equiv q$. The result being that no two deterministic programs are comparable but nonequivalent. Q.E.D.

This theorem shows that the deterministic programs (actually their equivalence classes) form an anti-chain. It might also be interesting to study the order of various layers of programs. Within this structure of layers, the deterministic programs seem to occupy the "middle" layers. Evidence for this is that both *skip* and *abort* are in the "middle" of the given prescriptions. In fact, *skip* is between *havoc* and *choose*. *Abort* is incomparable with all these prescriptions except for *miracle* and *bot*. Obviously there is a connection between the order, this theorem, and the "duality" based on conjugates noted in Chapter 3.

4.1.2.2.3. Correspondence with Informal Semantics. Three aspects of the informal semantics must be stressed by the formal version: (1) the precise statement of the meaning of the abstraction order, (2) evidence that a complete distributive lattice is induced by the order, and (3) evidence that the top and bottom are consistent with the interpretation given.

All these requirements are met. First, the abstraction order is defined in terms of the monotonic predicate transformers which are in turn defined in terms of implication in a first-order logic. Secondly, the structure induced by the order on a language is transferred from the structure on the predicate transformers, so given the language of Chapter 3 (similar to that used in the ARM) a complete distributive lattice is formed. Third, the top and bottom are *bot* and *miracle*, which have the meaning stated in the informal semantics.

4.2. Syntactic Development.

The intended semantics of the last section are an attempt to provide intuition and justification for the structure of the Abstraction Refinement Model. This structure is derived from the Laws of Programming discussed in Chapter 3. To avoid the traps of syntax, a "semantic" structure is developed by forming equivalence classes of programs under the abstraction order. This semantic structure simplifies definition of the syntactic transformations as well as the proofs of the algebraic properties of the structure. The transformations defined here are used in Chapter 5 to describe software evolution.

4.2.1. Construction of Semantic Structure. The syntactic structure of the ARM is given by the pre-ordered language $\langle \mathcal{L}, \sqsubseteq \rangle$. At the semantic level the properties of the language and order are easier to identify and exploit. This section defines the structure and proves that it is a complete distributive lattice (which corresponds to the intended semantics).

The transition into the semantic structure is made by forming the equivalence classes of programs with the abstraction order. The equivalence class of $p \in \mathcal{L}$ is

$$[p] = \{q \in \mathcal{L} | \; p \equiv q\} \,.$$

The collection of these classes is $\mathcal{L}/\equiv$ and is denoted $A^{\sqcup}$.

The operators on $A^{\sqcup}$ that are used here are the "meet" and "join" on equivalence classes. These are defined as

$$[a] \wedge [b] = [a \sqcap b]$$
$$[a] \vee [b] = [a \sqcup b]$$

for $a, b \in \mathcal{L}$. The operators are also well-defined:

LEMMA 4.7. *If* $[a_1] = [a_2]$ *and* $[b_1] = [b_2]$ *then* $[a_1 \sqcup b_1] = [a_2 \sqcup b_2]$ *and* $[a_1 \sqcap b_1] = [a_2 \sqcap b_2]$.

PROOF: Let $[a_1] = [a_2]$ and $[b_1] = [b_2]$, then $a_1 \equiv a_2$ and $b_1 \equiv b_2$. Consider the first result. By definition $a_1 \sqcap b_1 \sqsubseteq a_1$ and $a_1 \sqcap b_1 \sqsubseteq b_1$. These with the above equivalences imply that $a_1 \sqcap b_1 \sqsubseteq a_2$ and $a_1 \sqcap b_1 \sqsubseteq b_2$. Taking the "meet" of these on both sides gives $(a_1 \sqcap b_1) \sqcap (a_1 \sqcap b_1) \sqsubseteq a_2 \sqcap b_2$ or $a_1 \sqcap b_1 \sqsubseteq a_2 \sqcap b_2$. A similar argument proves $a_1 \sqcap b_1 \sqsupseteq a_2 \sqcap b_2$, so $a_1 \sqcap b_1 \equiv a_2 \sqcap b_2$. A dual argument

shows that $a_1 \sqcup b_1 \equiv a_2 \sqcup b_2$, and together these imply that $[a_1 \sqcap b_1] = [a_2 \sqcap b_2]$ and $[a_1 \sqcup b_1] = [a_2 \sqcup b_2]$. Q.E.D.

Although not as important as the meet and join, a sequential composition operator can also be defined on $A^{\sqcup}$. Define for $p, g \in \mathcal{L}$, $[p]; [q] = [p; q]$. That ";" is well-defined must be shown.

LEMMA 4.8. *If* $[a_1] = [a_2]$ *and* $[b_1] = [b_2]$ *then* $[a_1; b_1] = [a_2; b_2]$.

PROOF: Let $[a_1] = [a_2]$ and $[b_1] = [b_2]$. Show $a_1; b_1 \equiv a_2; b_2$. By assumption $a_1 \equiv a_2$ and $b_1 \equiv b_2$. By monotonicity $a_1; b_1 \sqsubseteq a_2; b_1$ and $a_1; b_1 \sqsupseteq a_2; b_1$, so $a_1; b_1 \equiv a_2; b_1$. Also by monotonicity, $a_2; b_1 \equiv a_2; b_2$. Using transitivity of the equivalence, these imply that $a_1; b_1 \equiv a_2; b_2$ as required. Q.E.D.

Both of the Lemmas indicate that monotonicity of the operators of the programming language allows their extension to operators on the equivalence classes. The implication is that operations at the syntactic level have corresponding operations at the semantic level. The abstraction order can also be extended to $A^{\sqcup}$.

Define the partial order $\leq$ by

$$[p] \leq [q] \quad \textit{iff} \quad p \sqsubseteq q \qquad \text{for } p, q \in \mathcal{L}.$$

This order can also be seen by noting that $[a] \wedge [b] = [a]$ whenever $a \sqsubseteq b$. This is easily shown since $[a] \wedge [b] = [a]$ if and only if $a \sqcap b \equiv a$, which holds whenever $a \sqsubseteq b$.

Using the definitions of the meet, join, sequential composition, the order and other operators, the properties of the language operators can be transferred to $A^{\sqcup}$. So $\wedge$ and $\vee$ are commutative, associative and idempotent. In addition to these properties others can be shown.

By letting $a, b \in \mathcal{L}$ such that $[a] \leq [b]$ and $[a] \geq [b]$, it can be seen that $\leq$ is anti-symmetric because $a \equiv b$, and $[a] = [b]$. This shows that the order $\leq$ is a partial order.

This increase to partial order status allows $\wedge$ and $\vee$ to become the meet and join of $A^{\sqcup}$, and because of this $\langle A^{\sqcup}, \wedge, \vee \rangle$ is a lattice.

The lattice $\langle A^{\sqcup}, \wedge, \vee \rangle$ is also complete and distributive. While completeness follows from the definition of $\top$ and $\bot$ in $\mathcal{L}$, the proof of distributivity requires the property of atomicity. To say that $A^{\sqcup}$ is *atomistic* means that each element of $A^{\sqcup}$ is a join of atoms [GRÄG78].

THEOREM 4.9. *$A^{\sqcup}$ is atomistic.*

PROOF: $A^{\sqcup}$ is the collection of equivalence classes of programs in the language $\mathcal{L}$. Recall how $\mathcal{L}$ is defined. Given a base deterministic language $\mathcal{L}_{Det}$, $\mathcal{L}$ is the closure of $\mathcal{L}_{Det}$ on the operators $\sqcup$ and $\sqcap$. The programs which correspond to the atoms in $A^{\sqcup}$ are the "meet" of several deterministic programs.

In $A^{\sqcup}$ each atom is equivalent to the meet of "deterministic" classes. Each deterministic class can then be written as the join of every atom to which it contributes. Because it is the bottom, $[\perp]$ can be written as the join of no atoms: $\bigvee \emptyset$. The rest of the classes can either be written as the join of atoms (hyper-deterministic classes) or the join of deterministic classes (non-deterministic). By definition, the non-deterministic classes can be written as the join of deterministic classes, and using the rewritten deterministic classes, can be written as the join of atoms. Q.E.D.

The property of distributivity is assured by the following theorem: A lattice L is distributive whenever $\forall x, y, z \in L\ (x \wedge y = x \wedge z\ \ \&\ \ x \vee y = x \vee z) \implies y = z$. The proof of distributivity of $A^{\sqcup}$ is based on this theorem and the property of atomicity.

THEOREM 4.10. *$\langle A^{\sqcup}, \wedge, \vee \rangle$ is distributive*

PROOF: Let $[x], [y], [z] \in A^{\sqcup}$. Suppose that $[x] \wedge [y] = [x] \wedge [z]$ and $[x] \vee [y] = [x] \vee [z]$, but that $[y] \neq [z]$. Since $A^{\sqcup}$ is atomistic $[x]$, $[y]$ and $[z]$ can be written as the join of atoms. So in order for $[y] \neq [z]$ there must be an atom $[a]$ such that $[a] \sqsubseteq [y]$ but $[a] \not\sqsubseteq [z]$. But it must also be that $[a] \not\sqsubseteq [x]$, because otherwise $[a]$ would be a disjunct of $[x] \wedge [y]$ but not of $[x] \wedge [z]$ and this contradicts the assumption that these are equal. However, if $[a] \not\sqsubseteq [x]$, then $[a]$ is a disjunct of $[x] \vee [y]$, but not of $[x] \vee [z]$. So $[x] \vee [y] \neq [x] \vee [z]$, which again contradicts the assumption. So by contradiction $[y] = [z]$, and $A^{\sqcup}$ is distributive. Q.E.D.

The relationship between $A^{\sqcup}$ and the intended semantics is of interest. Had the semantics been approached in a different way (perhaps denotational), showing their correspondence would be a matter of showing an isomorphism between $A^{\sqcup}$ and the corresponding structure in the semantics. However, the axiomatic approach to semantics does not provide for such an exercise.

The objective is to show that the structure induced on the language $\mathcal{L}$ by the order of the semantics is the same as that induced by the abstraction order of the Laws of Programming. Since both orders induce a complete, distributive lattice on the equivalence classes of programs, all that

remains to be shown is that the orders are identical. Note that these orders are denoted by the same symbol ($\sqsubseteq$). To distinguish them in the following, the order from the semantics is shown as $\sqsubseteq_S$ and that from the Laws of Programming as $\sqsubseteq_L$. That these orders are identical is easy to see.

For programs $p, q \in \mathcal{L}$, both orders relate $p \sqsubseteq q$ to the non-deterministic language operators in the same way:

$$p \sqsubseteq_L q \quad \textit{iff} \quad p \sqcup q \equiv_L q$$
$$p \sqsubseteq_L q \quad \textit{iff} \quad p \sqcap q \equiv_L p$$

and

$$p \sqsubseteq_S q \quad \textit{iff} \quad p \sqcup q \equiv_S q$$
$$p \sqsubseteq_S q \quad \textit{iff} \quad p \sqcap q \equiv_S p$$

In both cases the non-deterministic operators correspond to the join and meet on the equivalence classes of programs. Clearly, $\mathcal{L}/\equiv_L$ and $\mathcal{L}/\equiv_S$ are not just isomorphic, they are equal.

4.2.2. Transformations. The ARM draws on two important ideas. The first of these is the algebraic structure of the language $\mathcal{L}$ given by the Laws of Programming. The second is the idea of transformations forming design decisions as in the Draco paradigm. This section defines the ARM representation of these design decisions as transformations.

To achieve this definition the language must be given a narrow-spectrum structure. This is done by first partitioning $A^{\sqcup}$ and then transferring this structure back on $\mathcal{L}$. The result is a series of sub-languages $\mathcal{L}_i \subset \mathcal{L}$ which form levels in the narrow-spectrum structure. The levels are the basis for defining the transformations. Each transformation has as domain and codomain two levels of the structure.

4.2.2.1. "Fracturing" the Language. In order to impose a narrow-spectrum structure on the language, $A^{\sqcup}$ is partitioned. This is done by rebuilding $A^{\sqcup}$ from the bottom up.

The "bottom" of $\mathcal{L}$ is the set of *atomic* programs (which are also infeasible). This set is defined as

$$A_{\mathcal{L}} = \{p \in \mathcal{L} | \perp \sqsubseteq p \;\&\; q \sqsubseteq p \implies p \equiv q \text{ or } q = \perp, q \in \mathcal{L}\}$$

The corresponding set of equivalence classes in $A^{\sqcup}$ is

$$\mathcal{A} = A_{\mathcal{L}}/\equiv = \{[a] \mid a \in A_{\mathcal{L}}\}.$$

and is also the set of atoms of $A^{\sqcup}$.

The goal is to rebuild $A^{\sqcup}$ by taking the joins of the atoms in $\mathcal{A}$. The property of atomicity shows that this reconstruction of $A^{\sqcup}$ is possible. However to be useful in the definition of the transformations, $A^{\sqcup}$ must also be partitioned into "abstraction levels." This is done by defining the following sets.

$$A_0^{\sqcup} = \{[\bot]\} = \bigvee \emptyset$$

$$A_1^{\sqcup} = \mathcal{A}$$

$$A_{i+1}^{\sqcup} = \{[a] \vee [b] \mid [a] \in A_i^{\sqcup} \ \& \ [b] \in \mathcal{A}\} - A_i^{\sqcup} \qquad \text{for } 1 < i \leq |\mathcal{A}| - 1$$

The union of these sets is $A^{\sqcup}$:

$$A^{\sqcup} = \bigcup_{i=0}^{|\mathcal{A}|} A_i^{\sqcup}$$

(This can be seen by noticing that for $i > 1$ each $A_i^{\sqcup}$ is the $\vee$-closure of the preceding level with $\mathcal{A}$, minus any classes which simplify via idempotence.)

The $A_i^{\sqcup}$ also form the partition of $A^{\sqcup}$ which is used to define the transformations. Before proving that the $A_i^{\sqcup}$ indeed form a partition of $A^{\sqcup}$, two properties of the $A_i^{\sqcup}$ with respect to the ordering must be shown. These essentially state that the $A_i^{\sqcup}$ are "levels of abstraction." The first being that each $A_i^{\sqcup}$ is an anti-chain (all non-equivalent elements are incomparable). The second that between two levels $A_i^{\sqcup}$ and $A_j^{\sqcup}$, $i < j$, only proper relationships hold (i.e. $[a_i] < [a_j]$ but $[a_i] \neq [a_j]$).

LEMMA 4.11. *Each pair of elements of $A_i^{\sqcup}$, for $0 \leq i \leq |\mathcal{A}|$, is either equal or incomparable.*

PROOF: The proof is by induction on i showing for each i that for $[a], [b] \in A_i^{\sqcup}$ if $[a] \leq [b]$ or $[a] \geq [b]$ then $[a] = [b]$.

First consider $i = 0$, for $A_0^{\sqcup}$ the Lemma is trivially true because there is only one element. So the base case is actually $i = 1$.

For $i = 1$, $A_1^{\sqcup} = \mathcal{A}$ which is constructed from $A_{\mathcal{L}}$. Let $p, q \in A_{\mathcal{L}}$. If $p \sqsubseteq q$ or $q \sqsubseteq p$, then $p \equiv q$ by the definition of $A_{\mathcal{L}}$, and the corresponding classes in $\mathcal{A}$ are equivalent: $[p] = [q]$. So the Lemma holds for $A_1^{\sqcup}$.

For the induction step assume that the Lemma is true for $A_i^{\sqcup}$ where $i < k$. It must be shown that it is also true for $A_k^{\sqcup}$.

Let $[p], [q] \in A_k^{\sqcup}$, such that $[p] \leq [q]$. By definition, two elements $[a], [c]$ of $A_{k-1}^{\sqcup}$ and two elements $[b], [d]$ of A exist such that $[p] = [a] \vee [b]$ and $[q] = [c] \vee [d]$. So $[a] \vee [b] \leq [c] \vee [d]$.

This holds because either $[a] \leq [c]$ and $[b] \leq [d]$, or $[a] \vee [b] \leq [c]$ (since $[a] \not\leq [d]$). By the induction hypothesis the first alternative must mean $[a] = [c]$ and $[b] = [d]$; so $[p] = [q]$. For the second, consider $[a] \vee [b] = [c]$. Since $[a] \vee [b] = [p]$, this means $[p] = [c] \in A_{k-1}^{\sqcup}$. But also $[p] \in A_k^{\sqcup}$, which is not possible by definition. So the equality cannot hold, and if the second alternative is true then $[a] \vee [b] < [c]$. However, this implies that $[a] < [c]$ which violates the induction hypothesis. Therefore, only the first alternative holds and so $[p] = [q]$. Q.E.D.

The second Lemma relates $A_i^{\sqcup}$ and $A_j^{\sqcup}$ $(i < j)$ and sets the foundation for proving the existence of the partition.

LEMMA 4.12.

(1) *For all* $[a] \in A_j^{\sqcup}$, *there is a* $[b] \in A_i^{\sqcup}$ $(0 \leq i < j \leq |A|)$ *such that* $[b] < [a]$.

(2) *For all* $[a] \in A_j^{\sqcup}$, *there is no* $[b] \in A_i^{\sqcup}$ $(0 \leq i < j \leq |A|)$ *such that* $[b] = [a]$.

PROOF:

(1) The proof is split into two parts: a proof for the case when $j = i + 1$, and then a proof that this case implies the general case.

First let $j = i + 1$, and prove (1) by induction. For the base case let $i = 0$. Then there is only one element of $A_0^{\sqcup}$, namely $[\perp]$, and so by definition $[\perp] < [a]$ for all $[a] \in A_1^{\sqcup}$.

In the inductive step assume the Lemma holds for all $j < k$ (where $j = i + 1$), show that it holds for $A_k^{\sqcup}$. Let $[p] \in A_k^{\sqcup}$, then there is an $[a] \in A_{k-1}^{\sqcup}$ and a $[b] \in A$ such that $[p] = [a] \vee [b]$. This implies that $[a] \leq [p]$. The equality can be eliminated by noting that $[a] = [p]$ is not allowed by the definition so $[a] \neq [p]$ and the order is proper.

For general $0 \leq i < j \leq |A|$, there is a sequence $A_i^{\sqcup}, A_{k_1}^{\sqcup}, \ldots, A_{k_n}^{\sqcup}, A_j^{\sqcup}$ such that $k_1 - i = 1$, $k_l - k_{l+1} = 1$ and $j - k_n = 1$. By the first case the Lemma holds for each consecutive pair in this sequence. Therefore, for all $[a_j] \in A_j^{\sqcup}$ there is a chain $[a_i] < [a_{k_1}] < \cdots < [a_{k_n}] < [a_j]$. So by transitivity $[a_i] < [a_j]$.

(2) The second part follows from the fact that the order in (1) is proper. Let $[p] \in A_i^{\sqcup}$ and

$[q] \in A_j^{\sqcup}$ such that $[p] = [q]$. By the first part there exists an $[r] \in A_i^{\sqcup}$ so that $[r] < [q]$. But since $[p] = [q]$, it is also the case that $[r] < [p]$. This violates Lemma 11 since both $[r]$ and $[p]$ are elements of $A_i^{\sqcup}$, and they are not equal.

Q.E.D.

That the $A_i^{\sqcup}$ form a partition can now be shown. The goal is to show that each pair of $A_i^{\sqcup}$ is disjoint.

THEOREM 4.13. *$A^{\sqcup}$ is partitioned by* $\left\{A_0^{\sqcup}, \ldots, A_{|A|}^{\sqcup}\right\}$.

PROOF: This result follows from the fact that $A^{\sqcup} = \bigcup_{i=0}^{|A|} A_i^{\sqcup}$, and part (2) of Lemma 12. The first implies coverage to the set $A^{\sqcup}$ by the family of subsets $A_i^{\sqcup}$, the last implies that $A_i^{\sqcup} \cap A_j^{\sqcup} = \emptyset$ for any pair of subsets such that $i \neq j$. Both of which are required of a partition of $A^{\sqcup}$. Q.E.D.

This partition can be imposed on the language by defining the sets:

$$\mathcal{L}_i = \{p \in \mathcal{L} |\ [p] \in A_i^{\sqcup}\}.$$

Since the $A_i^{\sqcup}$ form a partition of $A^{\sqcup}$, the $\mathcal{L}_i$ form a partition of $\mathcal{L}$.

The partition of $\mathcal{L}$ gives it a "narrow-spectrum" structure with each $\mathcal{L}_i$ acting as a level of abstraction. These levels are the sets which act as the domain and codomain of the transformations.

4.2.2.2. Transformation Definition. The design decisions that the ARM tries to capture act on the syntactic form of a system description, but are motivated by semantics, i.e. the transformations are syntactic but are related to the semantics (the abstraction order). A number of different types of transformations correspond to actions which might be taken on system descriptions.

A set of basic transformations is included in the model. These are mappings $\mathcal{L}_i \xrightarrow{f} \mathcal{L}_j$ where $|i - j| \leq 1$. Note that if f maps p into q, then no element separates them (i.e. there is no r such that $p \sqsubseteq r \sqsubseteq q$ or $p \sqsupseteq r \sqsupseteq q$) unless $p \equiv q$ (only possible if $i = j$). Transformations which map $\mathcal{L}_i$ into $\mathcal{L}_i$ are *horizontal*, and all others *vertical*. Because of their nature most basic transformations (possibly excluding some horizontal ones) take infinitesimal steps in the lattice. More effective transformations which take much larger steps by "skipping" levels of abstraction can be formed by the composition of basic transformations.

Transformations which correspond to refinement map from a domain of higher abstraction to a codomain of lower abstraction. These *forward* transformations are mappings from $\mathcal{L}_i$ into $\mathcal{L}_j$ such

that $i > j$. Similarly there are *reverse* transformations which map from lower levels of abstraction to higher ones. These are defined as mappings from $\mathcal{L}_i$ into $\mathcal{L}_j$ such that $i < j$. Neither forward nor reverse transformations necessarily have an *inverse*.

Notice that for transformations such as the forward $f : \mathcal{L}_{i+1} \rightarrow \mathcal{L}_i$ it is not always the case that for all $p \in \mathcal{L}_{i+1}, f(p) \sqsubseteq p$. The dual can be said for the reverse transformation $g : \mathcal{L}_j \rightarrow \mathcal{L}_{j+1}$. A transformation $f : \mathcal{L}_i \rightarrow \mathcal{L}_j$ with the property that if $p \in \mathcal{L}_i$ then either $f(p) \sqsubseteq p$ (if f is forward) or $f(p) \sqsupseteq p$ (if f is reverse) is called *correctness-preserving*.

Correctness-preserving transformations are those which are desired in the manipulation of system descriptions because they allow the construction of a correct system from its specifications. Horizontal correctness-preserving transformations, in particular, correspond to identity maps in $A^{\sqcup}$, and legal (or sound) manipulations of system descriptions using algebraic laws (i.e. The Laws of Programming). Non-correctness-preserving transformations (called *degenerate*) correspond to erroneous actions on system descriptions which result in an incorrect system.

As noted composition can be used to build more effective transformations from the basic ones. The properties of correctness-preserving and degenerate transformations with respect to composition are outlined in the following claims.

Let $f : \mathcal{L}_i \rightarrow \mathcal{L}_j$ and $g : \mathcal{L}_k \rightarrow \mathcal{L}_i$ be either both forward or reverse transformations.

(1) If both f and g are degenerate, then $f \circ g$ can be correctness preserving.

(2) If f is degenerate and g is correctness-preserving, then $f \circ g$ is degenerate.

(3) If f is correctness-preserving and g is degenerate, then $f \circ g$ can be correctness-preserving.

(4) If both f and g are correctness-preserving, then $f \circ g$ is as well.

The first claim is justified by the observation that it is possible for f to correct the error made by g. For example, if $p \sqsubseteq q$ but r is not related to either, then if g maps p to r and f maps r to q, then the composition maps p to q. The second and fourth are fairly obvious, but the third is not. To see that this could be, consider p and q which are incomparable but have a meet r. The transformation $p \rightarrow q$ is degenerate, but $q \rightarrow r$ is correctness-preserving. Their composition is the same as $p \rightarrow r$, which is also correctness-preserving.

The first and third claims have implications for design decisions in software evolution. Neither claim states a principle to be encouraged. Both state that it may be possible to recover from a mistake. In one case by making another, and the other case by simply continuing as if no mistake

is made.

4.3. Summary.

The development of the Abstraction Refinement Model provides insight into the structure of the software evolution space. Using a wide-spectrum language like the non-deterministic language used here, the equivalence classes of programs under the abstraction order form a complete distributive lattice. Both the syntax and semantics exhibit these properties which allow the "layering" of the language to define syntactic transformations. While this layering imparts a "narrow-spectrum" structure to the language, this structure is not necessarily like that of a collection of narrow-spectrum languages used in practice.

Chapter 5. Software Evolution Process Characterization

Enabled by the structure and mechanisms of the Abstraction Refinement Model, the task remains to show how the software evolution process is represented by the ARM. The characterization to be described comes from the view that design decisions act upon system descriptions to achieve a refined system description. This view corresponds to that of the Draco Paradigm, and the transformation-based paradigm as exemplified by PROSPECTRA. Within this view, the transformations defined in the last chapter represent the actions carried out by design decisions.

An important realization is that these transformations act upon complete system descriptions to produce a new (complete) system description that "satisfies" the original. A transformation may perform some decomposition (functional or hierarchical) of the system, but does not produce a component separated from its context within the system. It is the composition of these transformations which form an action on the system.

Typically, actions on a software system are categorized either as development or maintenance. This separation has been motivated primarily by organizational views with little theoretical grounding. In the following, this separation is used to appeal to the familiarity of these notions, but then development and maintenance are combined to show that this separation is not inherent in software evolution.

The first goal for the ARM is to demonstrate that the separation between development and maintenance is "artificial;" doing so makes it obvious that the connection between them has existed. An understanding of how development and maintenance are characterized by the scheme allows us to see where the connections are made and the consequent influences of development on maintenance. These connections are made through the system descriptions.

In Chapter 4 the system descriptions are expressed in a non-deterministic language. Now, the system descriptions are considered to be more general, and could be documentation as well as program text. This assumption is reasonable if some restrictions are placed on the content and format of these documents.

The system descriptions are generally assumed to contain the information "necessary" for refinement to a lower level of abstraction. What is "necessary" depends on the semantics of the languages being used, but also varies with organizational, and project requirements. Thus, no fixed notion of

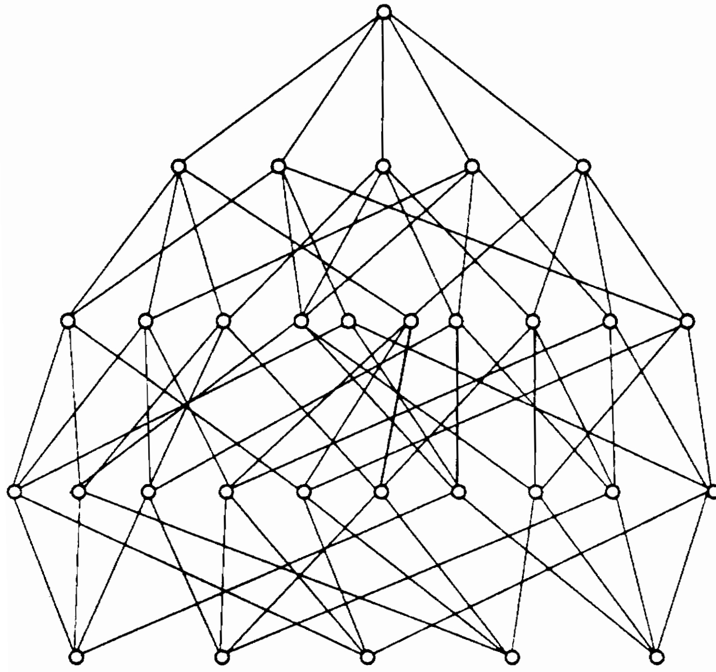


Figure 5.1. Illustration of the Lattice Structure of the ARM

"necessary" can be stipulated without loss of generality (imposing a methodological slant).

The structure of the equivalence classes of system descriptions ($A^{\sqcup}$) is a complete distributive lattice. Part of this structure is shown in Figure 5.1. In fact, what is shown is the "feasible" portion of a small sub-lattice of $A^{\sqcup}$. Despite the attention paid to infeasible system descriptions in Chapter 3, they are ignored here. So all diagrams show the structure as a variant on this join-semilattice. As mentioned, the diagrams actually show the structure of $A^{\sqcup}$. This is sufficiently complex to discourage expansion of the equivalence classes of system descriptions to show $\mathcal{L}$.

Although, a system description is not a complete set of documentation, it is complete in the sense that it defines a level of abstraction. In contrast, a complete set of documentation contains all the documentation from the highest to lowest defined levels. This documentation could contain each system description and all design decisions made (and alternatives considered). In some sense, this documentation is the context for the decisions made later in the process, particularly for those at a more concrete level (implementation or low-level design).

The *descriptive context* of a system is the collective documentation consisting of system descriptions and design decisions. A context is represented by a *path*, which is defined as a sequence of system descriptions connected by transformations. A path in which all transformations are correctness-preserving corresponds to a chain in the language structure. This notion of "context" is similar to the structure of the "Integrated Life-Cycle Model" [YAUS89].

5.1. Development.

The development process as characterized by the ARM is primarily a forward engineering activity. Resulting from development is an executable (hopefully) software system within a context built by refinement from an initial system description (in DoD-STD-2167 terminology: the System Requirements Specification [DSSD85]). Thus, the major product of development, usually considered to be just the software system, is the context.

Within the model, development has two phases: an analysis phase and a transformational phase. This is shown in Figure 5.2, which depicts the analysis phase acting upon the "top" part of the structure and the transformational phase, upon the lower part.

5.1.1. Analysis Phase. The analysis phase consists of those activities which lead to the synthesis of a single system description. These activities may include refinement of a single description or the combination of several descriptions into one. The operations used may include those from the

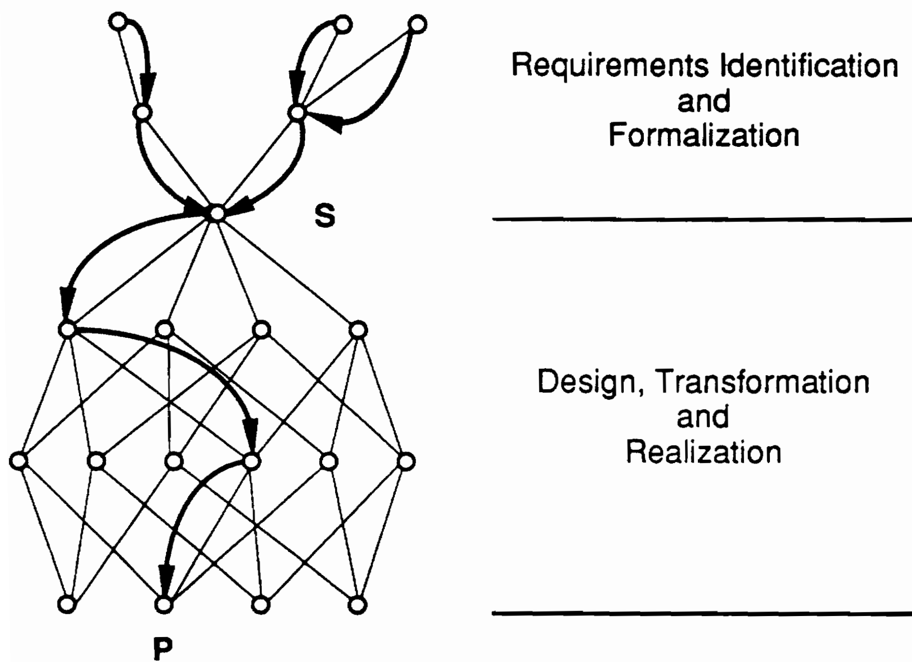


Figure 5.2. The Analysis and Transformational Phases of Development

language, forward transformations, or the meet operator.

The meet operator is used to combine several vague descriptions which contain (at least part of) the desired system description. The result is a partial description that can either be further refined or combined with other partial specifications to form the whole. The use of the meet operator during the analysis phase is represented as an inverted tree in Figure 5.2. The "root" of this tree is the ("initial") system description from which development continues. This description, call it s, helps constrain the solution space as any valid realization p must be a descendant ($p \sqsubseteq s$). The realizations of s correspond to the *ideal* generated by s. The sets of realizations for system descriptions s and p are shown in Figure 5.3.

Language operators are used to combine partial descriptions into larger (possibly still partial) system descriptions. For example, a case analysis might lead to the composition of several alternative system descriptions by the alternation operator. A hierarchical decomposition might be achieved by the sequential composition of several partial system descriptions, which are derived independently then composed. These might also be introduced through a design decision.

The language operators (mostly binary) are horizontal transformations, meaning that the result is not related to the operands. This is depicted for sequential composition in Figure 5.4(a). As mentioned above the application of the meet operator combines two system descriptions and appears as an inverted tree (Figure 5.4(b)). The forward transformations appear simply as downward paths (Figure 5.4(c)).

The descriptive context of the analysis phase can be very complex. It contains all system descriptions, partial and otherwise, which contribute to the definition of the initial system description. This context is important only to the transformational phase, although the initial system description must be included in the context of the transformational phase as well.

5.1.2. Transformational Phase. Development proceeds from the initial system description using a methodology proscribed by the objectives of the development project (i.e. desired qualities of the product)[ARTJ86]. A methodology consists of a set of methods which are modeled in the ARM as transformations. The goal of applying these transformations is to develop a system realization. This is shown simplistically in Figure 5.2, however development may also require some reverse engineering. Both forms of development are discussed below.

5.1.2.1. Context Construction. Each of the basic transformations takes an infinitesimal step toward achieving the final system realization. To achieve higher granularity, the composition of

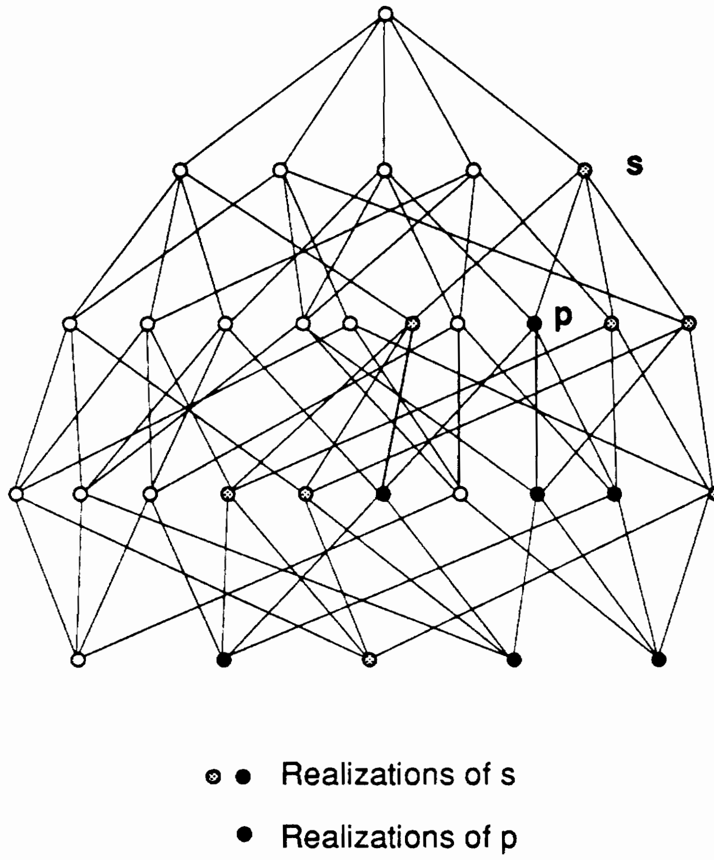
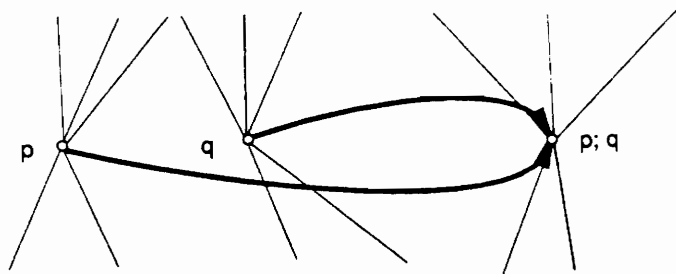
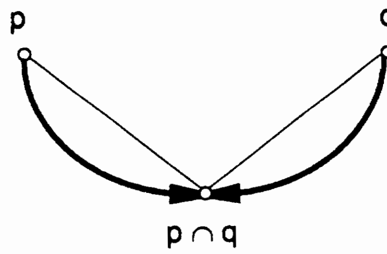


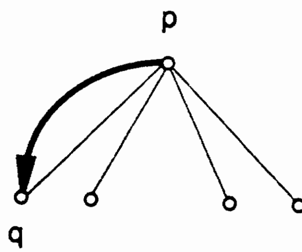
Figure 5.3. The Realizations of System Descriptions s and p.



(a) Sequential Composition Operator



(b) Meet Operator



(c) Forward Transformation

Figure 5.4. Operators and Transformations.

these transformations is necessary. Together with the system descriptions, these compound transformations form the context which terminates with a particular realization of the initial system description. Although the transformational part of the context is like a "tool" for building the system realization, the knowledge of the developer is required to construct and use this tool. The combination of tool and knowledge allows the implementation to be found. This is consistent with the work of Neighbors [NEIJ80], in which Draco is defined.

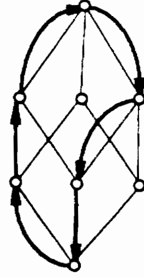
During the transformational phase the context is built by the addition of the design decisions and system descriptions. Each forward transformation adds to the context, so each transformation shown in Figure 5.2 contributes to the descriptive context of the system p.

5.1.2.2. Iterative Redefinition. The depiction of the transformational and analysis phases in Figure 5.2 is admittedly simplistic. Development methodologies such as rapid prototyping suggest an iterative definition/prototyping activity during the early stages of development. This can be characterized as well.

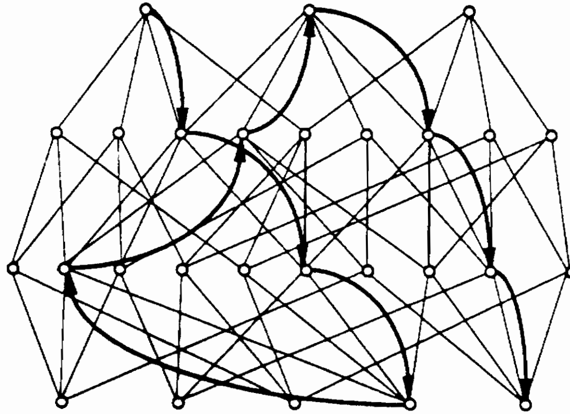
To model such an iterative process reverse transformations must be applied. The transformations take the current system description back to a level sufficient for the modification desired. These transformations generally do not result in closed loops, as such a path indicates that effort is being duplicated (Figure 5.5(a)). Instead, open loops are formed, representing the addition or change in the description (Figure 5.5(b)).

A reverse transformation reverses the effect on the context of a forward transformation. A reverse transformation *expands* the context by removing design decisions while a forward transformation *narrows* the context by adding them. This is suggested by Figure 5.3, where a move from p to s increases the number of possible realizations. In contrast, a move from s to p drastically reduces the possible realizations. However, the effect of a reverse transformation is not so simple.

In addition to removing the design decisions made below its target descriptions, a reverse transformation affects the context above as well. This is because the context is a plan of how the system can be realized from the initial system description. So, the context must be a monotonic chain from the original description to the current realization. Therefore, the effect of a reverse transformation is to realign the context upward until the new context coincides with the old. This coincidence begins with a system description which is a common abstraction of the source and target of the transformation and is an element of the context. Such system descriptions are called *common context elements*. The new context consists of design decisions and system descriptions not derived



(a) Ineffective Redefinition



(b) Effective Redefinition

Figure 5.5. Iterative Redefinition of Specifications.

in a top-down manner from the initial system description. Instead, these contextual objects are introduced by linking the new system description to the old context. (This is discussed further in the following section.)

5.2. Maintenance.

The ARM characterization of maintenance similarly has two phases. During the analysis phase the objective of the maintenance activity is identified. In the model this objective is assumed to be represented by an object q which is located in the lattice. Thus, q is in some sense the target object of the maintenance activity. Ironically, this assumption says that the goal of maintenance is in hand before starting, a much stronger assumption than we "know what we want."

The goal of maintenance is affected by its form. The four forms of maintenance are adaptive, perfective, preventive, and corrective [SWAE76]. Adaptive maintenance is indicated by changes in requirements, while the others are improvements within the established requirements. Each of the others has an objective of improving certain qualities. Corrective and preventive maintenance are aimed respectively at correctness and reliability, while perfective maintenance is targeted at the qualities of maintainability, reusability, testability, portability, and adaptability. The characterization of the maintenance forms in the ARM, however, is most affected by their relationship to changes in requirements.

Notice the parallel between the forms of maintenance and the types of degeneracy. Nonconformance implies the need for corrective maintenance (and perhaps preventive), and invalidity implies the need for adaptive maintenance. This relationship illustrates the utility of the ARM in motivating the need for maintenance.

5.2.1. Analysis Phase. While the analysis phase representation is less descriptive in the ARM, it is still worthy of discussion. Prior to the initiation of the transformational phase, certain information must be obtained. The analysis phase is intended to discover the information needed to guide a maintenance activity.

Four activities have been identified for the analysis phase:

(1) problem identification,

(2) objective determination,

(3) alternative solution evaluation, and

(4) solution selection [NANR90].

These activities are intended to develop a plan for the transformation which actually effects the realization of the objective software system.

5.2.2. Transformational Phase. Following the identification of the objective in the analysis phase of maintenance, the transformational phase begins. The transformations used in maintenance are both forward and reverse, but in contrast to the transformational phase of development, neither dominates the process. Maintenance requires balanced forward and reverse engineering efforts. The current realization p is the object of transformation, similar to the role of the original system description during development.

Figure 5.6 illustrates the maintenance process. Arrows pointing upward are reverse transformations; arrows pointing downward are forward transformations. The model does not prove instructive with regard to how the phases of maintenance are achieved, and the phases are not represented.

Notice that maintenance is not characterized above as performing a transformation directly from the current to the target realization. This is, however, possible both in the model and reality. Such a transformation is difficult and error-prone for most tasks and systems. The reason being that it does not exploit higher-level system descriptions to define the modification. From a programmer's view this amounts to a change made with only knowledge of the current and target realizations (but not their specifications). Although cognizance of this possibility is important, it is neither as interesting or complex as that presented here.

5.2.2.1. Executing Transformations. The maintenance form has an influence over the way in which activities are performed and depicted. In particular, the adaptive form differs greatly from the others. This is due to the nature of the change. While adaptive maintenance is driven by a change in requirements, the others are driven by the need for system quality improvement within the current requirements.

5.2.2.1.1. Maintenance Within Fixed Requirements. The perfective, preventive and corrective forms of maintenance are all performed with respect to a fixed initial system description (requirements specification). Therefore, the objective of maintenance is a realization of the current specifications with the proviso that the scope of the change might be restricted. Adaptive maintenance, discussed in the next section, differs.

Transformation begins with reverse transformations from the original specification p, and ends with forward transformation to the objective q. The reverse transformations comprise the process of *reverse engineering*, in which the information conveyed by the text of the realization is abstracted

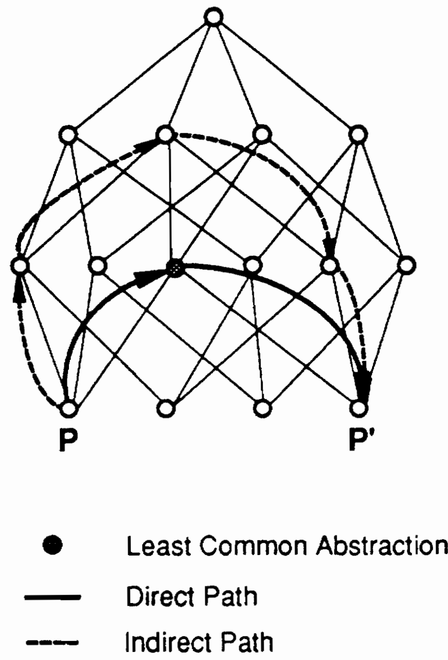


Figure 5.6. ARM Characterization of Software Maintenance.

to a form that subsumes the specification of the maintenance objective and the current realization. This corresponds to the elimination of design decisions and system descriptions so that the context is appropriate for the choices which lead to the revised system.

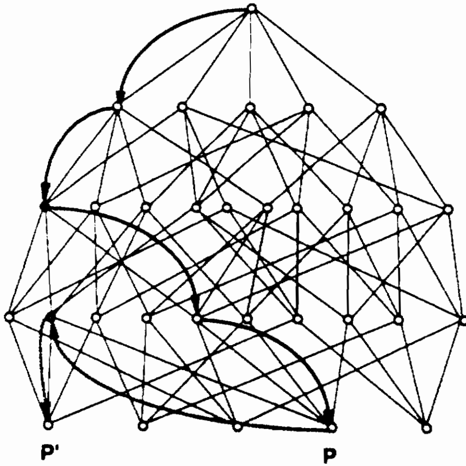
The system description α achieved by reverse engineering is a common abstraction of p and q since both $p \sqsubseteq \alpha$ and $q \sqsubseteq \alpha$. The poset can contain a number of common ancestors of both p and q , but the target of course is the "closest." The closest common ancestor is called the Least Common Abstraction (LCA) of p and q and is the choice (or "join") of the two: $p \sqcup q$ (the ***least upper bound*** or ***supremum***). Because it is the *least* common abstraction, the LCA is closest to both p and q by the number of intermediate realizations (or the length of the chains linking p to α, and α to q). (Note that the language structure does not have unique joins, so the LCA is in fact only one of a number of equivalent descriptions.)

Achievement of the least common abstraction also implies the possibility of the least amount of work in redefining the system context. The closer the common abstraction to the existing realization, the closer it is likely to be to the common context element. The new context passes through the common abstraction (since the forward engineering builds from this system description), and a shorter connection to the original context indicates less work. The shortest connection originates with the Least Common Context Element. Figure 5.7 shows the possibilities given an existing context and varying maintenance objectives.

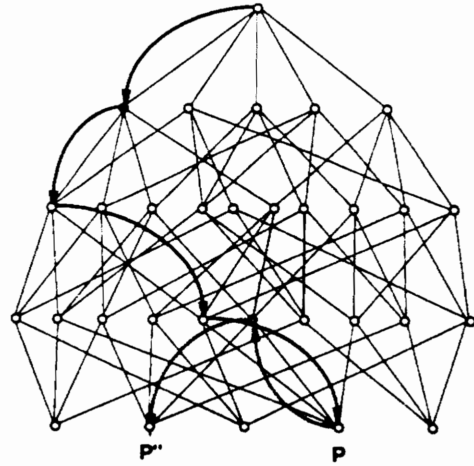
Clearly the distance from the objective and the least common context element depends on the objective being a realization of a context element. This is shown best by Figure 5.7(a) where the LCCE is a nearby specification, and 5.7(c) where the LCCE and LCA coincide. In the other parts of the figure, the LCCE is a higher level specification. This has implications for maintenance forms which are discussed later.

Since (basic) correctness-preserving transformations follow these chains, permitting only one per link (a pair of elements a, b such that $a \sqsupseteq b$ or $a \sqsubseteq b$), the length of these chains represents the amount of work to be done to achieve a common ancestor. Obviously, a maintenance path through the LCA is the shortest path attainable by reverse engineering and is the most desirable. This path is called an *direct path*, and is obtained by *direct path reverse engineering*. A path which passes through an ancestor which is not the LCA is called an *indirect path*.

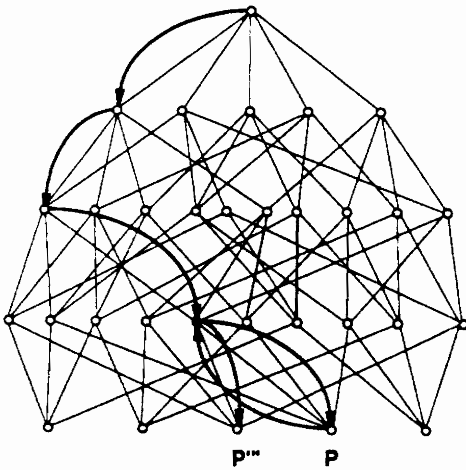
Once a common ancestor (preferably the LCA) has been reached through the reverse transformations, forward transformations are used to reach q. These transformations follow a path between



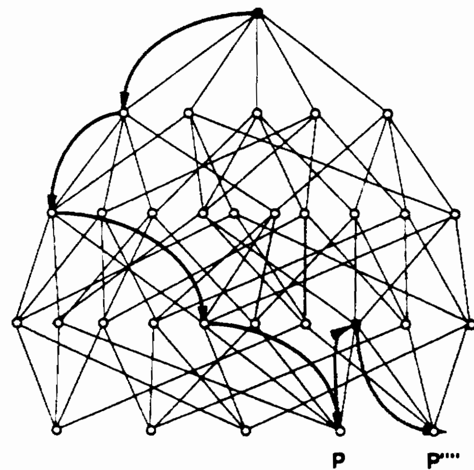
(a)



(b)



(c)



(d)

Figure 5.7. Relationship of the Context and the LCA (spotted) and the LCCE (black).

the LCA and q in a similar fashion to that of development transformations leading to a realization. Once q is reached, if it is a realization then the process stops; otherwise, a realization of q must be found by further forward transformations.

5.2.2.1.2. Maintenance with Changing Requirements. Adaptive maintenance is performed in response to a change in the initial system description. This form represents the changes to a system which are most closely related to development. The change in requirements may merely result in the need for developing a new system component or may require major changes to existing code.

To be effective the changes to the requirements must be of a certain nature. Primarily, the change must be restricted so that the current system is not a realization of the new requirements. This implies two things with respect to the ARM: (1) the new requirements cannot specify the old, and (2) the new requirements should not share the current system as a realization. While these restrictions seem obvious, they are not as clear in diagram form.

An example of adaptive maintenance is given in Figure 5.8. The original specification s, with realization p, is changed to s', with realization p' (it can be checked that no system description at this level other than s' meets the requirements of an effective change from s). Unlike the other forms, the new and the old context do not coincide, so no common context element exists.

The adaptive modification might not be as drastic as the depiction suggests. More than likely such a change leaves a majority of the context intact, with changes to some portions but mostly major additions. Despite this fact, how the changes could be made is still unclear.

One way of viewing the modification is as a mapping from the old context to the new (as suggested by Yau [YAUS84]). For some components of context elements, this mapping would be an isomorphism, and would primarily be the identity map. For other system descriptions, the mapping either drastically changes the structure of the component or adds a new one.

Relating to the other forms, a scheme for performing an adaptive modification is to find the LCA of the specifications by reverse engineering. Then the new specification and context could be found by forward transformations on the affected components. Here the LCA of the specifications serves the same role as the least common context element for the other forms of maintenance.

5.2.2.2. Issues in Achieving a Common Abstraction. The original conception of the ARM (as influenced mostly by Draco) is concerned with the variety of paths which lead to a common abstraction [NANR89]. This concern is motivated by a need to understand the reverse engineering

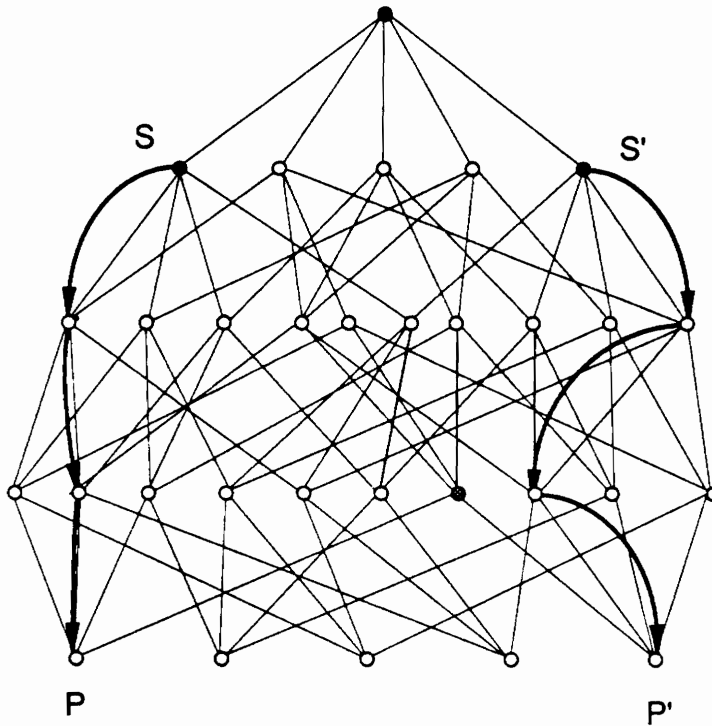


Figure 5.8. An Adaptive Maintenance Modification from p to p' initiated by changing requirement s to s'.

process. "Reverse engineering" is used here more generally than is usually the case. It typically means the completion of the context from the realization (a bottom up definition of the context) when the context is not complete. One definition is that reverse engineering comprises those "activities having the objective of assimilating information to enable effective and efficient system maintenance" [NANR89]. Here the term "reverse engineering" is used to mean that the information required for maintenance is obtained in some way, either by simply looking in the context, or by reconstructing the context.

If the context for a system is complete, the search for a common abstraction is possible without reconstruction. This is the ideal reverse engineering process, where all needed information is available [NANR89]. While this situation is possible, it is not always the case that the least common abstraction is within the context, although for all forms of maintenance a common context element exists.

A strong inverse relationship is evident between the level of abstraction of the least common context element (LCCE) and the forms of maintenance. This relationship is not empirically substantiated, but is shown by Figure 5.9. This figure depicts the relationship as a distribution function of the probability that a maintenance activity of a particular form has an LCCE at a certain level or lower. This shows the highest point in the context which is required to perform the maintenance activity [NANR89]. Although the LCCE is a common abstraction, it is unlikely to be the least common abstraction (Figure 5.7).

When the context is not completely available, "classical" reverse engineering is required to reconstruct it. However, the reconstruction of the actual context used in development is rarely possible; because forward transformations force abstraction into concreteness. Any application domain knowledge contained in higher level system descriptions is sure to be lost as development progresses. This precise, domain-specific knowledge is nearly impossible to recover.

The difficulty in finding the inverse transformations is best shown by analogy (given first in [NANR89]). Define division over the natural numbers as

$$a, b \in N \quad a \div b = \lfloor a/b \rfloor$$

Notice that a/b is a rational number and is in some sense "more abstract" than both a and b. Division for many algebras has multiplication as an inverse. For example

$$(4 \div 5) \times 5 = 4$$

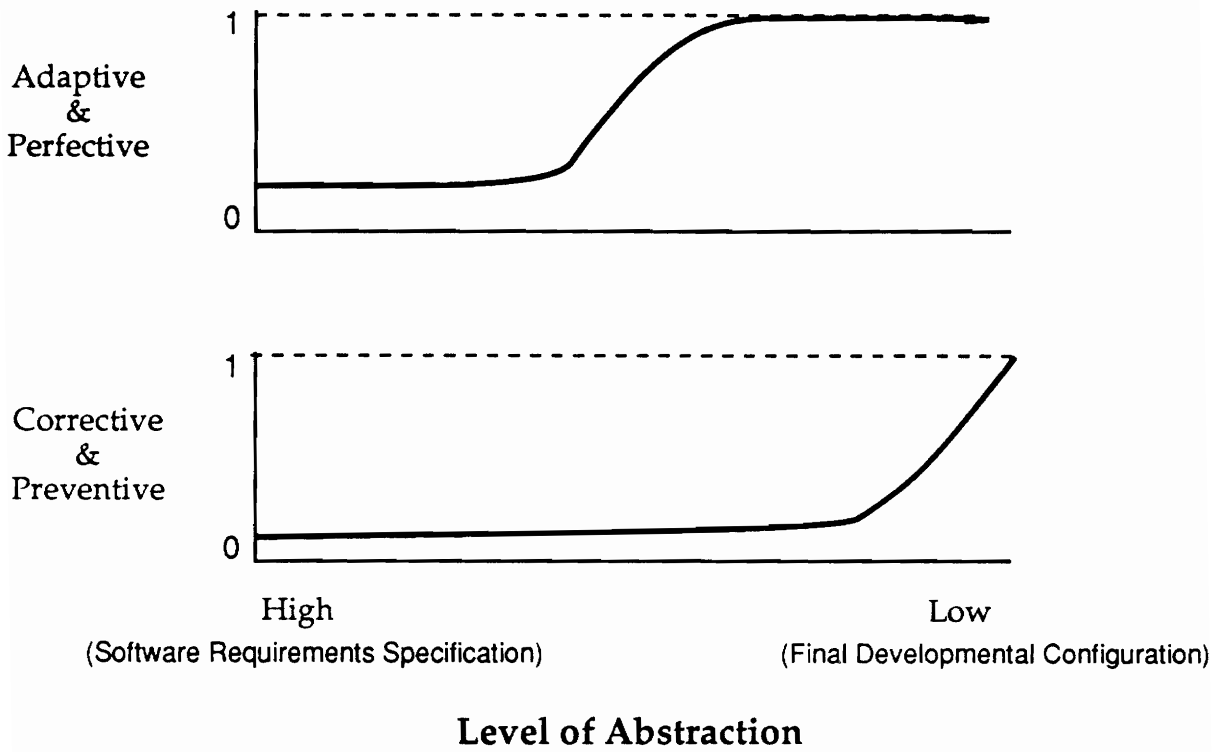


Figure 5.9. Illustrative Distribution Functions of the Abstraction Level of the LCCE for Maintenance Forms.

However, with division as defined for natural numbers

$$(4 \div 5) = \lfloor 4/5 \rfloor = 0$$

and

$$(4 \div 5) \times 5 = 0 \times 5 = 0 \neq 4$$

Therefore, multiplication is unable to recover the information lost by application of the floor operator. An inverse would have to "know" something about the original operands to recover this information.

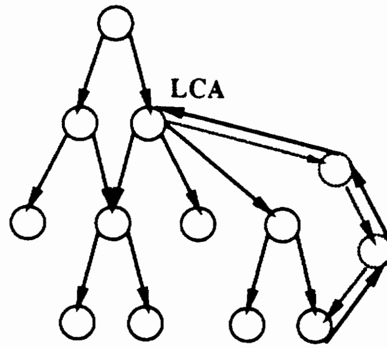
The application of the floor operator, in the above example, is forced by the requirement that the quotient be a natural number. The operations illustrate the loss of information about the rational intermediate result. By forcing an object of one type into an object of a simpler type, such as rational into natural numbers or specifications into design, some information is lost about the original form. The implication is that, functionally, the forward transformations are onto but not one-to-one, and therefore not invertible [NANR89].

The inability to find an exact inverse, motivates the search for an approximation. The reverse transformations comprising this approximation constitute a *shadow path* [NANR89]. Shadow paths take three forms. The first such path goes directly to the least common abstraction (an *extended* shadow path), the second to a common abstraction which is an ancestor of the LCA (an *indirect* shadow path), and third to a common ancestor unrelated to the least common ancestor (an *oblique* shadow path). These forms are shown in Draco-like diagrams in Figure 5.10.

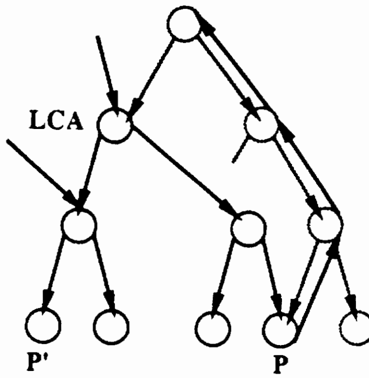
Assuming correctness-preserving transformations. only the indirect type of shadow path is admissible due to the structure of the language. To prove this it must be shown that extended shadow paths coincide with direct paths, and that oblique paths cannot exist. For the extended paths it suffices to show that all paths from p to $p \sqcup q$ are of the same length (i.e. require the same number of basic transformations). For oblique paths it is necessary to show their existence contradicts that $A^{\sqcup}$ is a lattice.

THEOREM 5.1. *All correctness-preserving shadow paths are indirect.*

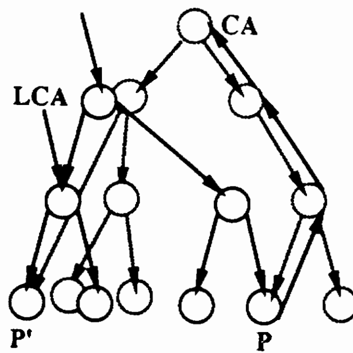
PROOF: It is first shown that oblique paths cannot exist. The definition of an oblique path requires that an $r \in \mathcal{L}$ exist so that $r \sqsupseteq p$ and $r \sqsupseteq q$ but $r \not\sqsupseteq p \sqcup q$. So r is an upper bound of p and q but is not a specification of $p \sqcup q$.



(a) Extended Shadow Path



(b) Indirect Shadow Path



(c) Oblique Shadow Path

Figure 5.10. Draco-style Depiction of Shadow Paths (from [NANR89, p. 31]).

Now switch to $A^{\sqcup}$. Either $[r]$ is not related at all to $[p] \vee [q]$ or $[p] \vee [q] \geq [r]$. In the first case, the existence of $[r]$ violates the uniqueness of the least upper bound. In the second, $[r]$ would have to be the least upper bound, and so violates the definition of the join.

Consider extended paths. It must be shown that every path from p to $p \sqcup q$ is of the same length. This is argued using atomicity. Suppose that q can be written as the join of n atoms $a_1, \ldots, a_n$. Then $p \sqcup q$ can be written as $p \sqcup a_1 \sqcup \cdots \sqcup a_n$. Each reverse basic transformation from p maps into $p \sqcup a$ for some atom a (these elements cover p). So there are n transformations from p to the $p \sqcup a_i$, and from each of those there are $n - 1$ transformations to the $p \sqcup a_i \sqcup a_j$. Eventually, the transformations will converge on $p \sqcup a_1 \sqcup \cdots \sqcup a_n$, each path taken being n basic transformations long. Therefore, each path from p to $p \sqcup q$ is of the same length. Q.E.D.

These results show that the impact of the information loss (problem) is not as complicating as might be thought. The restriction of possibilities (given correctness-preserving transformations) to either exact or indirect paths promotes better structure in finding a common abstraction. For degenerate transformations, however, no guarantees can be made.

In summary, maintenance consists of reverse and forward engineering efforts which are balanced on each side of a common abstraction. Reverse engineering poses the biggest problem in dealing with maintenance effectively due to the difficulty in finding a common abstraction, which can be further complicated by the information loss problem.

5.3. Complete Life-Cycle.

The relationships between development and maintenance become most striking with the ARM. Both development and maintenance consist of an analysis and transformational phase and even share transformations. Success in the maintenance process is dependent upon the quality of the products of the development process. This claim is justified by the fact that the context of the existing system is the context for any maintenance activities.

From the earlier discussion, development begins with a high-level system description and ends with a low-level realization. Maintenance on the other hand begins with a low-level realization and ends with another low-level realization. Together development and maintenance form a path which traces down through the higher levels of the structure and then hops around at the lower levels. An example is depicted in Figure 5.11.

Figure 5.11 shows the software evolution process as a path through the language structure (the

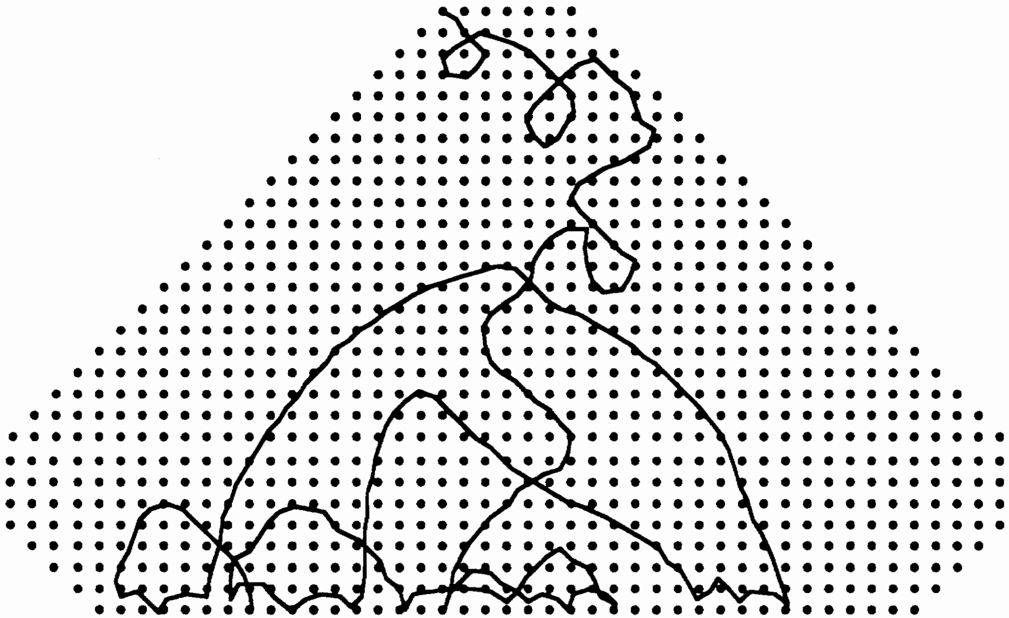


Figure 5.11. Evolution Path.

evolution space). The separation between development and maintenance is not striking, though clearly there is a distinction. This difference is significant enough to imply the insufficiency of development methods for maintenance, but the impact of development on maintenance is unquestionable.

The importance of the connection between development and maintenance is given in terms of the system context. An ideal maintenance process is desirable to reduce the effort for maintenance activities. To achieve this, a more complete context is required, and the first possible construction of this context is in development. The lack of a development context requires construction of a context at the earliest application of maintenance. Similarly, the context induced by maintenance activities is required for future maintenance [NANR89].

Chapter 6. Conclusions

The Abstraction Refinement Model developed in previous chapters has very simple elements: (1) a collection of system descriptions, (2) transformations or mappings between system descriptions, and (3) an abstraction order on the collection of system descriptions. Through the use of the Laws of Programming and an intended semantics, these components combine to form an algebraic structure which describes the space of software evolution. The contributions of this model and its utility are explored below. First, the contributions of the model are evaluated in terms of the criteria stated in Chapter 1. Secondly, the utility of the model is critiqued.

6.1. Evaluation of Model Contributions.

Chapter 1 states some very specific goals for the ARM. These are that it should relate maintenance to development, be non-methodological, and represent imperfections. The extent to which these criteria are met by the ARM is explored in the following.

6.1.1. Relate Maintenance to Development. This goal is directed at avoiding the relative neglect of maintenance by life-cycle models. Although they can be characterized in the ARM, neither maintenance nor development is an explicit part of the model. This fact helps the model to meet this goal.

The characterizations of maintenance and development given in Chapter 5 show them as activities which build and rebuild a system context. The relationship is clear; development builds a system within its descriptive context, and maintenance departs from this context. The implication is that maintenance is dependent on development for the context, this context being needed initially as an information source and object of transformation during maintenance. An incomplete context impacts the ability to perform the maintenance, either requiring a blind transformation from current to target systems or requiring reverse engineering to reconstruct the context.

While this relationship between development and maintenance underscores the importance of planning for maintenance during development, the problem is not restricted to development. The model shows that maintenance, even at a low level, impacts higher levels of the context. Without realigning the context to represent changes, its usefulness is lost. For systems with a long life, failure to realign the context may have more drastic consequences than not recording the context during development because of the problems related to having inconsistencies in the documentation.

6.1.2. Non-Methodological. A critical issue in representing software evolution is to avoid methodological bias since a model should not restrict the possibilities of actions on systems. Although the ARM allows representation of all actions on the system descriptions, it has theoretical and practical implications for methodologies.

The ARM is considered non-methodological since all transformations on system descriptions are included in the model. Because of this generality, no restrictions are placed on the representation of actions on a system. For example, a non-correctness preserving transformation may be applied although a good methodology might prevent this.

A methodology can, in fact, be characterized in terms of the ARM as a selection of transformations. For each system description, the methodology provides one or more transformations to carry out each desired action on that system description. This relates well with the characterization of a methodology based on the OPA framework [ARTJ86], where the objectives of a project imply the application of certain principles which correspond to selections of methods or transformations. Despite the independence of the model from methodologies, the reverse is not the case. Methodologies can and should derive instructional guidance, primarily in the formation of principles, from the ARM.

Another way in which methodologies are affected is in the order of tasks or applications of methods (transformations). While a methodology might allow tasks to be performed in arbitrary order, the ARM suggests that certain transformations have "levels of applicability," that impose a theoretical order on the application of methods. The ARM in this case restricts the collection of possible methodologies.

Other implications come from the characterization of software evolution given in Chapter 5. The most important of these regard the descriptive context. The ARM makes no requirement for recording either system descriptions or transformations, and does not require that maintenance be performed using the context. However, both have implications on the effectiveness of a methodology, and use of the ARM helps identify these. The ARM is currently being exploited in this capacity in the definition of a maintenance methodology for the AEGIS combat system (see [NANR90]).

6.1.3. Recognize Imperfections. The representation of deficiencies in the software evolution process is an issue of generality. The goal is to provide the means for motivating the need for maintenance and maintenance methodologies. The latter has been met to some extent as discussed above, but the former is still open.

The only deficiency discussed is degeneracy of both system descriptions and transformations. However, degeneracy relates only to the quality of correctness. While correctness pertains to both adaptive and corrective forms of maintenance, the impact on the other forms is not obvious. Perfective maintenance, in particular, is directed at improving qualities other than correctness and reliability. These qualities are not related to correctness, and so the notion of degeneracy is not sufficient to represent the lack of these qualities. Reliability is related to correctness, and therefore the need for preventive maintenance can be justified to some extent by degeneracy.

The question is whether or not these neglected qualities are crucial to the representation of imperfection. Certainly they are desirable qualities, but none is worth having without correctness. It therefore seems immediately sufficient to represent imperfections in terms of correctness.

6.2. Extensions for Model Improvement.

The last section suggests that the Abstraction Refinement Model successfully meets the goals for a model of software evolution from Chapter 1, in part because of the simplistic nature of the model which provides sufficient utility for most of the goals. However, some of the assumptions and the approach to defining the ARM may have resulted in a loss of generality.

This section addresses extensions to the model which might help regain that utility. The two extensions discussed affect the "language" on which the model is based. The first is extending the language to the full specification language discussed by Hoare et al. [HOAC87], and the second concerns the representation of narrow-spectrum languages.

6.2.1. Hoare's Notion of Specification Language. Although the ARM is based on the Laws of Programming enunciated by Hoare et al. [HOAC87], that work goes further with the definition of the specification language. Here the specification language is simply a non-deterministic language extended by the dual of non-deterministic choice, but Hoare's language includes non-implementable operations.

Two questions are raised by this extension. Is the language with non-implementable operators truly more general than that without? And, if so, how do programs constructed with these operators relate to themselves and other programs via the abstraction order?

The first question is really one which asks for the relationship between infeasibility and non-implementability. Certainly infeasibility is a component of non-implementability, but the existence of another component is not immediately clear. One possibility may relate to non-computability, but

that is a property of problems rather than specifications and so it is not clear what the corresponding property may be. It may be infeasibility, and so the question is mute.

The answer to the second question is dependent on the answer to the first. If the non-implementable operators add something to the language, then the abstraction order must also be extended to cover the new elements. Otherwise, if non-implementability is simply infeasibility, then no real extension of the order is required since these new elements correspond to old ones.

6.2.2. Narrow-spectrum Issues. The ARM as currently defined is based on a wide-spectrum language. However, it is more likely that narrow-spectrum languages will be used in practice. A move to narrow-spectrum languages introduces some new problems in the development of the ARM.

If the ARM is a good representation of the software evolution space, then this structure should be present even for a system of narrow-spectrum languages. Essentially, this would mean that the narrow-spectrum languages are embeddable in the language of the ARM. Note that if the language with non-implementable operations is more general, then the embedding may be into the extended language instead. Perhaps this extension would be a better model, since it might be the case that a narrow-spectrum language has an element which has no realization in a lower language.

The definition of an ARM over a system of narrow-spectrum languages would require a a study of the structure of systems of languages. A *language system* includes a collection of languages, each with an abstraction order defined on it, and a collection of inter-language abstraction orders between the languages. The orders should be so that the inner- and inter-language orders are transitive with respect to each other. While any possible collection of languages is allowed by this definition, the relationships among the languages would imply the usefulness of a language system.

In practice, the languages of the system would be ordered by abstraction. A way to accomplish this is to define an order on languages which indicates that every element of the "lesser" language is the realization of some element of the "greater" language. A language system with a sequence of languages ordered this way would have a "target" language where all implementations occur. The system would in some sense be *effective* since an element of the target language would be a realization of some element of every other language, and therefore has a specification.

The understanding of the language relationships and the corresponding language system classifications are necessary for the study of an ARM defined on narrow-spectrum languages. The variety of properties of language systems implies that the corresponding Abstraction Refinement Model's have varying utility. This might prove useful in developing requirements on the language systems

used in practice.

Bibliography

[ANTP87] P. Antonini, P. Benedusi, G. Cantone, and A. Cinitile, *Maintenance and Reverse Engineering: Low-Level Design Documents Production and Improvement*, in "Proceedings of the Conference on Software Maintenance – 1987," IEEE Computer Society Press, 1987, pp. 91–100.

[APTK86] K. R. Apt and G. D. Plotkin, *Countable Non-determinism and Random Assignment*, Journal of the ACM **33** 4 (October 1986), 724–767.

[ARAG85] G. Arango, I. Baxter, and P. Freeman, *Maintenance and Porting of Software by Design Recovery*, in "Proceedings of the Conference on Software Maintenance – 1985," IEEE Computer Society Press, 1985, pp. 42–49.

[ARTJ86] J.D. Arthur, R.E. Nance, and S.M. Henry, "A Procedural Approach to Evaluation Software Development Methodologies: the Foundation," Technical Report SRC-86-008, Systems Research Center, VPI&SU, September 1986.

[BACC88] C. Bachman, *A CASE for Reverse Engineering*, Datamation **34** 13 (July 1, 1988), 49–56.

[BACR89] R.J.R. Back and J. von Wright, *A Lattice-Theoretical Basis for a Specification Language*, in "Mathematics of Program Construction: Proceedings of the International Conference," LNCS#375, ed. J.L.A. van de Snepscheut, Springer-Verlag, Berlin, 1989, pp. 139–156.

[BAKA86] A.L. Baker, J.M. Bieman, D. Gustafson, and A. Melton, "Modeling and Measuring the Software Development Process," Report TR-CS-86-5, Department of Computing and Information Sciences, Kansas State University, November 1986.

[BARJ89] J. Barwise, *Mathematical Proofs of Computer System Correctness*, Notices of the American Mathematical Society **36** 7 (1989), 844–851.

[BELJ77] J. Bell, and M. Machover, "A Course in Mathematical Logic," North-Holland, Amsterdam, 1977.

[BENH56] H.D. Benington, *Production of Large Computer Programs*, in "Proceedings of ONR Symposium on Advanced Programming Methods for Digital Computers," Office of Naval Research Symposium Report ACR–15, 1956.

[BERJ84] J.A. Bergstra, and J.W. Klop, *Algebraic Tools for System Construction*, in "Logics of Programs," LNCS#164, eds. E. Clarke and D. Kozen, Springer-Verlag, Berlin, 1984, pp. 34–44.

[BIRG40] G. Birkhoff, "Lattice Theory," 6th Edition 1984, American Mathematical Society, Providence, Rhode Island, 1940.

[BOEB76] B.W. Boehm, *Software Engineering*, IEEE Transactions on Computers **C-25** 12 (December 1976), 1226–1241.

[BOEB86] B.W. Boehm, *A Spiral Model of Software Development and Enhancement*, ACM Software Engineering Notes **11** 4 (August 1986), 14–24.

[BROM86] M. Broy, A. Geser, and H. Hussman, *Towards Advanced Programming Environments Based on Algebraic Concepts*, in "Advanced Programming Environments: An International Conference," LNCS#244, eds. R. Conradi, T. Didriksen, and D. H. Wanvik, Springer-Verlag, Berlin, 1986, pp. 455–470.

[BROS84] S. D. Brookes, *A Semantics and Proof System for Communicating Processes*, in "Logics of Programs," LNCS#164, eds. E. Clarke and D. Kozen, Springer-Verlag, Berlin, 1984, pp. 68–85.

[CHAN88] N. Chapin, *Software Maintenance Life Cycle*, in "Proceedings of the Conference on Software Maintenance - 1988," IEEE Computer Society Press, 1988, pp. 6–13.

[CLIW81] W.D. Clinger, "Foundations of Actor Semantics," PhD dissertation, MIT, 1981.

[DIJE76] E.W. Dijkstra, "A Discipline of Programming," Prentice-Hall, Englewood Cliffs, NJ, 1976.

[DIJE90] E.W. Dijkstra, C.S. Scholten, "Predicate Calculus and Program Semantics," Springer-Verlag, New York, 1990.

[DOWM86] M. Dowson, *The Structure of the Software Process*, ACM Software Engineering Notes **11** 4 (August 1986), 6–8.

[DSSD85] "Dod-STD-2167A, Defense System Software Development," United States Department of Defense, June 4, 1985.

[FETJ88] J. H. Fetzer, *Program Verification: the Very Idea*, Communications of the ACM **31** 9 (1988), 1048–1063.

[FREP77] P. Freeman, *The Nature of Design*, in "A Tutorial on Software Design Techniques, Second edition," IEEE Computer Society Press, 1977, pp. 29–36.

[GAMS88] S. A. Gamalel-Din, and L.J. Osterweil, *New Perspectives on Software Maintenance Processes*, in "Proceedings of the Conference on Software Maintenance – 1988," IEEE Computer Society Press, 1988, pp. 14–22.

[GIDR84] R.V. Giddings, *Accommodating Uncertainty in Software Design*, Communications of the ACM **27** 5 (May 1984), 428–343.

[GRÄG78] G. Grätzer, "General Lattice Theory," Birkhauser, Stuttgart, 1978.

[HOAC78] C.A.R. Hoare, *Communicating Sequential Processes*, Communications of the ACM **21** (1978), 666–677.

[HOAC85] C.A.R. Hoare, *The Mathematics of Programming*, in "Foundations of Software Technology and Theoretical Computer Science," LNCS #206, ed. S. N. Maheshwari, Springer-Verlag, New York, 1985, pp. 1–18.

[HOAC87] C.A.R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin, *Laws of Programming*, Communications of the ACM **30** 8 (August 1987), 672–686.

[HOAC89] C.A.R. Hoare, *The Varieties of Programming Languages*, in "TAPSOFT '89," vol 1, LNCS#352, eds. J. Diaz, F Orejas, Springer-Verlag, Berlin, 1989, pp. 1–18.

[JONC80] C. B. Jones, "Software Development: A Rigorous Approach," Prentice-Hall International, London, 1980.

[KELS88] S. E. Keller, *Grammar-Based Program Transformation*, in "Proceedings of the Conference on Software Maintenance – 1988," IEEE Computer Society Press, 1988, pp. 110-117.

[KRIB88] B. Krieg-Brückner, *Algebraic Formalisation of Program Development by Transformation*, in "ESOP '88: Proceedings of the Second European Symposium on Programming," LNCS#300, ed. H. Ganzinger, Springer-Verlag, Berlin, 1988, pp. 34–48.

[KRIB89] B. Krieg-Brückner, *Algebraic Specification and Functionals for Transformational Program and Meta Program Development*, in "TAPSOFT '89," vol 2, LNCS#352, ed. J. Diaz, F Orejas, Springer-Verlag, Berlin, 1989, pp. 36–59.

[LEHM84] M.M. Lehman, *A Further Model of Coherent Programming Processes*, in "Proceedings of the Software Process Workshop," IEEE Computer Society Press, 1984, pp. 27-35.

[LEHM85] M. M. Lehman, and L. A. Belady (eds.), "Program Evolution: Process of Software Change," Academic Press, 1985.

[MAIM89] M.G. Main, and D.L. Black, "Semantic Models for Total Correctness and Fairness," Technical Report CU-CS-417-88, Department of Computer Science, University of Colorado, March 1989.

[MILA86] A. Mili, J. Desharnais, and J. Gagne, *Formal Models of Stepwise Refinements of Programs*, ACM Computing Surveys **18** 3 (Sept 1986), 231–276.

[MILR80] R. Milner, "A Calculus of Communicating Systems," LNCS#92, Springer-Verlag, Berlin, 1980.

[MORC88] C. Morgan, *The Specification Statement*, ACM Transactions on Programming Languages and Systems **10** 3 (July 1988), 403–419.

[MORJ89] J.M. Morris, *Laws of Data Refinement*, Acta Informatica **26** 4 (February 1989), 287–308.

[MUSJ87] J.D. Musa, A. Iannino, K. Okumoto, "Software Reliability: Measurement, Prediction, Application," McGraw-Hill Book Co., New York, 1987.

[NANR89] R. E. Nance, B. J. Keller, D. Boldery, "Documentation Production Under Next Generation Technologies," Technical Report SRC-89-001, Systems Research Center, VPI&SU, 15 January 1989.

[NANR90] R. E. Nance, J.D. Arthur, B. J. Keller, "Requirements for a Software Maintenance Methodology," Technical Report SRC-90-001, Systems Research Center, VPI&SU, 18 January 1990.

[NEIJ80] J. M. Neighbors, "Software Constructions Using Components.," PhD Dissertation, University of California, Irvine, 1980.

[NEIJ84] J. M. Neighbors, *The Draco Approach to Constructing Software from Reusable Components*, IEEE Transactions on Software Engineering **SE-10** 5 (September 1984), 564–574.

[NELG89] G. Nelson, *A Generalization of Dijkstra's Calculus*, ACM Transactions on Programming Languages and Systems **11** 4 (October 1989), 517–561.

[OVEC88] C.M. Overstreet, J. Chen, and F. Byrum, *Program Maintenance by Safe Transformations*, in "Proceedings of the Conference on Software Maintenance – 1988," IEEE Computer Society Press, 1988, pp. 118–123.

[PETL87] L. Peters, "Advanced Structured Analysis and Design," Prentice- Hall, 1987.

[PLOG76] G.D. Plotkin, *A Powerdomain Construction*, SIAM Journal of Computing **5** 3 (September 1976), 452–487.

[PLOG80] G.D. Plotkin, *Dijkstra's Predicate Transformers and Smyths Powerdomains*, in "Abstract Software Specifications," LNCS#86, ed. D. Bjørner, Springer-Verlag, Berlin, 1980, pp. 527–534.

[SCHD86] D.A. Schmidt, "Denotational Semantics: A Methodology for Language Development," Allyn and Bacon, 1986.

[SCHN87] N. F. Schneidewind, *The State of Software Maintenance*, IEEE Transactions on Software Engineering **SE-13** 3 (March 1987), 303–310.

[SHAM84] M. Shaw, *Abstraction Techniques in Modern Programming Languages*, IEEE Software **1** 4 (October 1984), 10–26.

[SMYM78] M.B. Smyth, *Power Domains*, Journal of Computer and System Sciences **16** (1978), 23–36.

[SNEH88] H.M. Sneed, and G. Jandrasics, *Inverse Transformations of Software from Code to Specification*, in "Proceedings of the Conference on Software Maintenance – 1988," IEEE Computer Society Press, 1988, pp. 102–109.

[SWAE90] E. Swanson, *The Dimensions of Maintenance*, in "Proceedings of the Second International Conference on Software Engineering," 1976, pp. 492–497.

[SWAE90] E. Swanson, and C. M. Beath, *Departmentalization in Software Development and Maintenance*, Communications of the ACM **33** 6 (June 1990), 658–667.

[TAUD89] D. Taubner, and W. Vogler, *Step Failures Semantics and a Complete Proof System*, Acta Informatica **27** (1989), 125–126.

[YAUS84] S. Yau, "Methodology for Software Maintenance," Final Technical Report RADC-TR83 -262, NTIS AD-A143-763/1, February 1984.

[YAUS88] S. Yau, R. Nicholl, J. Tsai, and S. Lin, *An Integrated Life-Cycle Model for Software Maintenance*, IEEE Transactions on Software Engineering **14** 8 (August 1988), 1128–1144.

[ZAVP89] P. Zave, *A Compositional Approach to Multiparadigm Programming*, IEEE Software **6** 5 (September 1989), 15-25.

Vita

Benjamin J. Keller was born to Raymond E. Keller and Cecile P. James on the sixth of July, 1964 in Iowa City, Iowa. He was raised in Blacksburg, Virginia and attended both private "alternative" and public schools in Blacksburg and Hollins, Virginia. His undergraduate education is from Western Kentucky University in Bowling Green, Kentucky. He holds a Bachelor of Science degree in Computer Science, and his degree also includes minors in Mathematics and Geology. His professional experience includes research in algorithms for artificial intelligence, issues in documentation generation, and software maintenance methodologies.