Strategies for Introducing Formal Methods into the ADA Life Cycle

J.A.N. Lee

Karl Nyberg

TR 88-9

Strategies for Introducing Formal Methods into the ADA Life Cycle

J.A.N. Lee
Virginia Tech

Karl Nyberg
Grebyn Corp.

TR 88-9

Also published by the Software Productivity Consortium as SPC-TR-88-002, January 1988.

The work reported herein was supported in part by the Virginia Center for Innovative Technology, Grant #SPC-00-002 and in part by the Software Productivity Consortium (Reston, VA).

Table of Contents

Section — Page

ABSTRACT 1

1.0 INTRODUCTION 1-1

1.1 BACKGROUND 1-1

1.2 THE NEED FOR VERIFICATION 1-6

1.3 THE ISSUES FOR RESOLUTION 1-7

1.4 FORMALISM IN THE LIFE CYCLE 1-10

1.5 CONVERTING THE LIFE CYCLE TO A FORMAL APPROACH .. 1-10

1.6 OVERVIEW OF THE REMAINDER OF THE REPORT 1-10

2.0 THE APPLICATION OF FORMALISMS TO THE LIFE CYCLE 2-1

2.1 OVERVIEW OF CURRENT APPROACHES TO FORMAL DESCRIPTION 2-4

2.1.1 A COORDINATED, COMPREHENSIVE FORMALISM FOR PROGRAM DEVELOPMENT 2-4

2.1.2 IMPLEMENTATION AND RUN-TIME CONSIDERATIONS 2-8

2.1.3 CONCURRENCY, EXCEPTIONS, AND FORMALISMS ... 2-9

2.2 PHASING FORMALISMS INTO THE LIFE CYCLE 2-10

2.2.1 FORMAL ASSISTANTS 2-15

3.0 CONCLUSIONS 3-1

4.0 RECOMMENDATIONS 4-1

5.0 ACKNOWLEDGEMENTS 5-1

BIBLIOGRAPHY 1

APPENDIX A: SURVEY OF FORMAL DEFINITION METHODS A-1

A.1 SOFTWARE LIFE CYCLE A-1

A.1.1 SOFTWARE QUALITY A-1

A.1.2 SOFTWARE DEVELOPMENT PARTICIPANTS A-1

A.2 ANALYSIS OF SPECIFICATION TECHNIQUES A-3

A.2.1 REQUIREMENTS FOR SPECIFICATION TECHNIQUES .. A-3

Table of Contents (Continued)

A.2.2 INTRODUCTION TO SPECIFICATION TECHNIQUES . . . A-3

A.2.3 APPLICABILITY OF SPECIFICATION TECHNIQUES A-5

A.3 THE MODEL FOR SOFTWARE DEVELOPMENT A-6

A.3.1 COMPARISON OF SPECIFICATION TECHNIQUES A-6

A.3.2 THE MODEL AND THE METHOD A-6

A.4 DISCUSSION A-9

A.5 BIBLIOGRAPHY A-10

A.6 OVERVIEWS OF FORMAL DESCRIPTION TECHNIQUES A-23

A.6.1 PETRI NETS AS A HIGH LEVEL SPECIFICATION TOOL A-23

A.6.1.1 GENERAL CONCEPTS A-23

A.6.1.2 CHARACTERISTICS OF PETRI NETS A-23

A.6.1.3 APPLICABILITY OF PETRI NETS A-23

A.6.1.4 A HIGH LEVEL SPECIFICATION TOOL A-26

A.6.1.5 BIBLIOGRAPHY -- PETRI NETS A-28

A.6.2 TEMPORAL LOGIC AS A HIGH LEVEL VERIFICATION TOOL A-30

A.6.2.1 GENERAL CONCEPTS A-30

A.6.2.2 CHARACTERISTICS OF TEMPORAL LOGIC . . . A-31

A.6.2.3 APPLICABILITY OF TEMPORAL LOGIC A-31

A.6.2.4 A HIGH LEVEL VERIFICATION TOOL A-31

A.6.2.5 BIBLIOGRAPHY -- TEMPORAL LOGIC A-32

A.6.3 THE AXIOMATIC APPROACH AS A LOW LEVEL VERIFICATION TOOL A-35

A.6.3.1 GENERAL CONCEPTS A-35

A.6.3.2 CHARACTERISTICS OF AXIOMATIC APPROACH A-36

A.6.3.3 APPLICABILITY OF AXIOMATIC APPROACH . A-36

A.6.3.4 A LOW LEVEL VERIFICATION TOOL A-37

Table of Contents (Continued)

A.6.3.5 BIBLIOGRAPHY -- AXIOMATIC APPROACH . . A-38

A.6.4 OPERATIONAL SEMANTICS AS A LOW LEVEL SPECIFICATION TOOL A-40

A.6.4.1 GENERAL CONCEPTS A-40

A.6.4.2 CHARACTERISTICS OF OPERATIONAL SEMANTICS A-40

A.6.4.3 APPLICABILITY OF THE OPERATIONAL APPROACH A-40

A.6.4.4 A LOW LEVEL SPECIFICATION TOOL A-41

A.6.4.5 BIBLIOGRAPHY -- OPERATIONAL SEMANTICS A-43

A.6.5 DENOTATIONAL APPROACH AS A SPECIFICATION AND VERIFICATION TOOL A-44

A.6.5.1 GENERAL CONCEPTS A-44

A.6.5.2 CHARACTERISTICS OF THE DENOTATIONAL APPROACH A-44

A.6.5.3 APPLICABILITY A-45

A.6.5.4 A SPECIFICATION AND VERIFICATION TOOL A-45

A.6.5.5 BIBLIOGRAPHY -- DENOTATIONAL SEMANTICS A-46

A.6.6 ALGEBRAIC APPROACH AS A SPECIFICATION AND TRANSFORMATION TOOL A-49

A.6.6.1 GENERAL CONCEPTS A-49

A.6.6.2 CHARACTERISTICS OF ALGEBRAIC APPROACH A-49

A.6.6.3 APPLICABILITY A-49

A.6.6.4 AN INTERMEDIATE LEVEL SPECIFICATION AND VERIFICATION TOOL A-49

A.6.6.5 BIBLIOGRAPHY -- ALGEBRAIC APPROACH . . A-50

APPENDIX B: A SURVEY OF POTENTIAL TOOLS B-1

B.1 PROGRAM DESIGN LANGUAGES B-1

Table of Contents (Continued)

B.1.1 ANNA B–1

B.1.2 PAMELA B–3

B.1.3 BYRON B–4

B.1.4 RAISE B–6

B.1.5 LARCH B–7

B.1.6 DISCUSSION B–8

B.1.7 BIBLIOGRAPHY -- PROGRAM DESIGN LANGUAGES .. B–9

B.2 FORMAL SOFTWARE ASSISTANTS B–10

B.2.1 AFFIRM B–10

B.2.2 PROLOG B–11

B.2.3 GYPSY B–11

B.2.4 SDVS (STATE DELTA VERIFICATION SYSTEM) B–12

B.2.5 INA JO B–12

B.2.6 VERUS B–13

B.2.7 ABEL B–13

B.2.8 EVES B–14

B.2.9 HDM B–14

B.2.10 VDM B–15

B.2.11 PVS B–15

B.2.12 BOYER-MOORE THEOREM PROVER B–16

B.2.13 MAVEN B–17

B.2.14 DISCUSSION B–19

B.2.15 BIBLIOGRAPHY -- FORMAL SOFTWARE ASSISTANTS B–20

List of Figures

Figure No.		Page
Figure 2–1:	Basic Element of a Life Cycle	2–1
Figure 2–2:	The Software Development Model	2–7
Figure 2–3:	Components of a Formal Life Cycle Model	2–13
Figure A–1:	Waterfall Model	A–2
Figure A–2:	The Software Development Model	A–8
Figure A–3:	Petri Net Examples	A–24
Figure A–4:	Dijkstra's Five Philosophers Problem	A–25
Figure A–5:	Parallelism Using Petri Nets	A–26
Figure A–6:	Control Structure	A–27
Figure A–7:	Rendezvous Mechanism	A–27

ABSTRACT

This is the final report of a short study of the applicability of formal definition techniques to program development activities with specific emphasis on using the programming language Ada.

The portion of the study reported here encompassed three elements:

- A review of the various formal definition techniques;
- A study of the existing and planned tools in programming development environments; and
- An examination of life cycle methodologies with the objective of inserting formalized techniques.

As a result of these activities, this report proposes that a plan for a formalized life cycle can be developed and that the initial steps should be:

1. The identification of the appropriate formalisms to be used at each stage of the life cycle and the interface requirements necessary to integrate the formalisms into the overall life cycle model;
2. The identification of the tools and environments which will serve as the carriers for the formal components; followed by
3. The implementation of a complete software development system.

1.0 INTRODUCTION

This is the final report of a short study of the applicability of formal definition techniques to program development activities with specific attention to using the programming language Ada as part of a larger "Ada Project" within the Software Productivity Consortium. Support for this project was supplied by the Center for Innovative Technology of the Commonwealth of Virginia and the Software Productivity Consortium. The portion of the study reported here encompassed three elements:

- A review of the various formal definition techniques that have been applied to elements of programming systems, with a view to investigating the potential for creating a coordinated approach to the complete formal description of the components of a program. Whereas in the past formal definition techniques have been applied singly to programming languages, this preliminary study envisages the use of a number of different techniques, each tailored to the element of the programming system where they have the closest relationship and where they can most accurately describe the semantics of that element.
- A study of the existing and planned tools in programming development environments, with a view to their use in:
    1. Supporting a formalized programming development activity,
    2. Providing the technological and computational support for program development with formalized verification, and
    3. Providing a bridge between current programming activities and the targeted fully formal system.
- An examination of the currently accepted life cycle methodologies with the objective of inserting formalized techniques into the definitional, transformational, and review processes.

1.1 BACKGROUND

The need to verify the correctness of programs has been recognized since the early programs were written by such pioneers as von Neumann [KNUTH] and Turing [DAVIS], who both accompanied their programs by formal proofs of correctness. As programming grew from a mathematical activity accompanied by proofs, through a coding activity that followed the verified designs completed by mathematicians, to a combined design and coding activity undertaken by programmers, the verification process was lost. In its place programmers substituted empirical testing as a means to raise the assurance level of the software. As the complexity of programs and systems grew, the ability to test satisfactorily has further diminished the confidence level in software developed in an ad hoc manner.

The emergence of software engineering as a discipline was as much predicated on the need to bring order from this chaos as it was a means for providing a methodology for increasing the quality of developed software. The use of various techniques (e.g., structured programming) and management styles (e.g., chief programmer team concept) combined with a systematic methodology (embedded in a so-called life cycle) has provided a first step in formalizing the development process. This has been achieved through two requirements:

1. The identification of clearly defined intermediate stages in the life cycle, accompanied by an audit trail of documentation, that raises the confidence level that the development process is disciplined and well managed; and
2. A series of reviews whereby the expertise of peers is brought to bear (hopefully) to identify errors and mistakes that are not then allowed to ripple through the remainder of the development activity.

While there is a plethora of such techniques and methods, and many of them have a basis in some formalism (such as software metrics), the majority of their applications are empirical. The software engineering process is at a stage that is akin to that of Civil Engineering in the mid-1800's when the processes of construction were being formalized, but the process of design was yet unable to provide complete proofs of correctness. Even in those days, some impressive feats of engineering were completed that have stood the test of time, but this may be more attributable to conservatism and high factors of safety than to good design.[1] In some respect we are beginning to see some elements of the same two qualities (conservatism and factors of safety) being applied to software. Conservatism in engineering can be seen as using only those materials and designs that have worked well previously -- the concept of reusability. It is much more difficult to apply factors of safety by "beefing up" software, but we are seeing the beginnings of this

[1] In a recently published biography of John von Neumann, by his brother Nicholas Vonneuman, a third brother (Michael) tells the story of one of his Professors of structural engineering saying:"... and the safety factors should be selected so that, if possible, the bridge should not collapse even once during its first five years of use." John commented: "Why of course, one can always secure 100% probability from satistical formulae, but then the spread of ranges and expenses become prohibitive." Interestingly enough there is a better analogy between these comments and software, due to its ability to recover from "catastrophic" failures, than with bridges!

concept in N-person programming, fault tolerant software and the use of strongly typed languages. Many of these less rigorous approaches have been built in to existing methodologies such as advocated by Yourdon, Jackson, etc. These approaches need not be abandoned; instead they may well be the basis of a system that is to be extended by formal systems.

This report looks at the longer term progression of change with a view to organizing the transition from a disciplined management approach to software development to a methodology that involves a great deal more formalism and that will eventually provide the same degree of confidence in the correctness of the product as in an engineering environment. This does not imply that the disciplined management approach (using the life cycle as its core) should be replaced by a totally formal, mathematical mechanism. Rather, as in engineering design and construction (and maintenance) the formal approach provides the tools and methods that are utilized within the existing development framework.

The Ada environment for program development is particularly interesting in this regard though it is strongly believed that the recommendations developed herein are equally applicable to other language based environments. In recent years there have been some initial excursions into this area both in terms of formal description of the language and its environment and tools that support program development [BJOR1], [BJOR2], [BJOR3], [BJOR4], [BJOR5]. The integration of these components into a methodology for program development that would support different intellectual levels of participation is an important part of this plan.

Traditional management approaches to software quality include IV&V (Independent Verification and Validation). In IV&V the word "verification" basically means checking between stages in the software development process to be sure that all items have been correctly handled. For example, that each of the points in the specification is accomplished by a portion of the design, that no functions exist in the design that are not explicitly required in the specifications, and that all the connections between the requirements specification and the design have been properly documented.

As an approach to increasing the discipline of development some methodologies have begun to use various metrics and models to predict the complexity and error proneness of large software developments. There are also formal procedures for software audits and so forth. These topics are well covered in many current software engineering text and guide books (e.g., [PFLEE]).

The life cycle methodology has been neglected by more formally oriented computer scientists because of its being generally a management technique. The Cleanroom [MILLS] and Software Safety [LEVES] approaches have their roots in these now classical approaches to software management.

These techniques should be adapted to use the emerging formal description methods, the automatic cross referencers between stages in program development, and the limited formal verification technologies that are now becoming available. As an intermediate step, formalized techniques for stepwise integration into the life cycle need to be developed with eventual verification in mind, and disciplined, systematic management techniques continue to be used to assure that the software is "visually" (statically) correct to the same extent as we achieve today.

Formal verification is the highest technological approach to increasing assurance in the correct functioning of computer software today. Unfortunately, it is viewed by many as the only means by that the highest level of assurance can be obtained. To such people, other approaches, such as testing, configuration management, development methodology, etc. may have their benefits and places, but only verification alone can provide absolute assurance. This has resulted in the unfortunate position that verification has become an all-or-nothing proposition for software development requiring stratospheric levels of assurance [PREST]. Such mentality has not served to bolster the application of formal verification technology. As we will note later, the need for formal verification belies the inability to place a high degree of reliance on the transformational development of objects in the life cycle; essentially verification is a backward looking step!

The formal verification process consists of preparing, prior to the development of software, the formal specification of a model of the intended behavior of the software. Some effort may be placed (as described previously) in the analysis of the specifications to ascertain their completeness and internal consistency. Then, following the software development methodology, designs and implementations at the various levels of the software are completed, and formal correspondence with the specification is performed, resulting in proofs of correctness of the implementation with respect to the specification. The specification can be written in a non-procedural transformation language, while the implementation is provided in a procedural implementation language. The formal proof of correctness consists of showing that the two separate, hopefully somewhat orthogonal, descriptions have a proper correspondence. That is, showing that the transformation process is correct. Research is still ongoing to find means by which requirements and specification, which are in narrative terms, can be formalized. Unless some means can be created to transform requirements into specifications and specifications into design, verifiably correct programs can be inconsistent with the requirements. The extraction of significant items from specifications and the constructions of a set of test criteria could form the basis of formalizing narrative specifications.

One of the benefits of formal verification technology is its formality. This is an area that seems to have varied amounts of support during the design of Ada. The syntax of the language can be described by an LALR(1) grammar, allowing the development of syntax parsers in as little as a few hours, given the current state of parser generator technology. In the early requirements documents for the Ada language, verification was mentioned as a desirable goal, but the language contains many constructs that prevent this goal. In order to conduct formal proofs of consistency between the specification and the implementation, a formal description of the language semantics is also necessary. Some effort has been undertaken by the European Economic Community (EEC) in this area, but it has not sufficiently matured to a stage where it can be utilized in formal verification. Even the only viable specification language for Ada, ANNA, has been geared more toward the utilization of run time assertion checks, rather than formal verification, and has largely ignored the aspects of parallelism. At least one known effort is involved in extending Anna to overcome these deficiencies. [PREST] has suggested that the completion of a verification environment suited to the needs of NSA trusted software requirements will require of the order of 30-60 man-years of effort. It is expected by the plan developed here that a start can be made into this methodological development that will produce results without waiting for the completion of the full verification environment. Furthermore, specification languages were viewed both from the perspective of their necessity for formal code verification and from the perspective of their applicability to other techniques, both formal and less formal, for increasing the understanding of software. Research in application of formal methods to Ada outside of the code verification domain was investigated also. Two specific areas are the application of formal methods to specification analysis and to runtime assertions. From that it was concluded that "Formality in software development should not be all or nothing." This report strongly agrees with this conclusion and suggests means by that a gradual introduction of formalisms into the life cycle can be achieved.

1.2 THE NEED FOR VERIFICATION

There are five "classical" methods of demonstrating various degrees of correctness of a program:

1. N-version programming;
2. Post development testing;
3. Recomputation of the solution by an independent method;
4. Substitution of the generated results into the requirements specification; or
5. Formal verification.

N-version programming [CHEN] and post development testing can be considered to be managerial approaches to correctness assurance that are performed during the life cycle and that are independent of the run-time environment. However neither of these methods yields a degree of confidence in the resulting software that is acceptable in "mission critical systems". Redundant computation of the solution by the use of an independent algorithm, and the substitution of results into a quantified version of the requirements, are both techniques that can be used to "instrument" the run-time program, and thus continue to assure correctness. Both of these methods impose a burden that affects the efficiency of the program, that may be intolerable in real-time systems. The National Computer Security Center (NCSC) has defined criteria to certify trusted software. Currently the highest designated level is "A1", at which level the specification or design must be proven to be correct by "formal methods". On the other hand the relationship between the specifications (or design) and the implemented code can be established informally. In this context, it is to be assumed that "informality" still requires a high level of discipline and structure, and is not entirely an ad hoc activity. NCSC also defines a level of software beyond A1 that requires formal verification throughout the development process.

The formal verification process consists of preparing, prior to the development of software, the formal specification of a model of the intended behavior of the software. Such requirements will require that verification of correctness (i.e., proofs) be founded on already well-founded descriptions and methods. That is, it will be impractical to start such a formal process from "scratch" for each system being developed. The tools for formal verification:

- A basic algebra or formalism,

- A description of the language of specification, design or implementation,
- A transformation process between successive elements of the development system (from specification to design, design to implementation, etc), and
- A theorem "prover",

need to be developed in advance, to be shown to be "trusted", and preferably to have been used elsewhere in operationally accepted systems. For example, the existence of a formal description of the programming language Ada is of little consequence to a program written in the language unless there is some means of "compiling" a formal description of the program that will be input to a system based on formal technology. Similarly, proofs of correctness are only viable if there are two ends to the proof:

- The conjecture (that the given program is correct), and
- The specification of correctness in terms of set of desired properties.

Both these elements need to be expressible in a common language, or at least a pair of languages between which there is a well-formed transformation.

Irrespective of the requirements for "trusted software" established by NCSC, the overall requirement for developing verified software surpasses the bounds of defense and strategic systems. The same processes of program development and verification should be as applicable to domestic systems as mission critical systems. However, the cost of complete verification may be beyond the means of domestic ventures. As stated by [PREST] "the application of this advanced technology requires such significant amounts of available computing resources as to dwarf other methods of increasing assurance." Moreover [PREST]: "the use of application of this technology will also increase the difficulty of project management."

1.3 THE ISSUES FOR RESOLUTION

As a guide to this study, the following issues were raised and are answered in the succeeding sections:

1. What is the objective/purpose of using formalisms?
2. How formal is formal?
3. Can a meaningful progression from the informal (disciplined procedural approach) to the formal (mathematical) be developed?
    a. Is the progression of transformation of processes continuous or discrete?

   b. Is there a hierarchy or structure to progressive formalism application?

   c. Does the formalism exist throughout the transition from procedural to mathematical approaches?

   d. If the eventual goal is a completely mathematically formal approach, can a first step towards that goal be meaningfully procedural?

   e. Would it be better to consider establishing a procedural approach that would then be overlayed with a mathematical formalism, or would the formalism replace the procedures?

   f. In an approach that uses the overlay methodology, does the procedural component stay the same or does it change as formalism is added to the overlay?

   g. Can certain elements of the procedural model be formalized independently of other elements? Or does the formalization have to take place in layers throughout the whole model?

   h. Is formalism just another "tool" to be used in conjunction with other models? What other choices exist?

   i. Does everything have to be formalized?

   j. Can procedural formalisms be developed that are mutable to mathematical formalisms?

4. Where in the life cycle can formalisms be applied?

   a. Can mathematical formalisms be melded in a life cycle (such as Software Development Specification MIL-STD-2167-A) or is it an additional layer?

   b. If so how?

5. Can formalism (procedural or mathematical) be assisted, guided, and/or imposed by tools?

   a. What tools currently exist at any level of formalism?

b. Can a section be added to each element of the guide that is concerned with formalisms, or is it a separate issue?

c. Can the formal definition of a programming language be used to create a check list of items to be investigated when using a particular language construct?

13. If so, to what effect?

14. What degree of training will be needed for those who will manage a formal development process, program within this domain, perform V&V in this domain, perform maintenance activities in the domain, or accept products from the domain?

15. What can be accomplished through formalisms that cannot be accomplished by other means?

1.4 FORMALISM IN THE LIFE CYCLE

If we are to accept the premise that the development of a verification system will require 30–60 man–years of effort, then we must also recognize that the insertion of that technology into the "software factory" will also require additional considerable effort. The transformation from a disciplined program development environment to a formal system cannot be accomplished instantaneously. Nor would it be appropriate to throw out existing informal methodologies and thus lose contact between intuitive (humanistic) concepts and the (automated) formal systems. Experience with the application of new technology in the past has taught us to use the human approach as a validation check for some time after the formal system has been installed.

1.5 CONVERTING THE LIFE CYCLE TO A FORMAL APPROACH

We anticipate that a completely (and solely) formal life cycle of program development is so far in the future as not to be worthy of consideration here, except as an ultimate concept. Thus it will be necessary to design a program development life cycle that encompasses the concepts of procedural quality assurance but that progessively supports more and more of the elements of the model by formal means.

1.6 OVERVIEW OF THE REMAINDER OF THE REPORT

The studies of formal description techniques (including a subjective evaluation of the applicability of the techniques to verification activities) and verification support tools, that

6. Can existing tools:

   a. Meet the requirements of this task?

   b. Be modified to accomplish the goals of introducing formalisms?

   c. Be combined in an environment to fulfill the goals that they cannot individually meet?

7. Is the formal definition of a programming language useful?

   a. To language processor implementors?

   b. To program developers?

   c. To verification activities?

   d. If not, why not?

   e. If so, how?

8. Is the formalism for a programming language (such as VDM) equally appropriate to:

   a. The description of programs written in the language?

   b. The process of writing programs in the language?

   c. The specification of processes for writing programs in the language?

9. Is the application of formalisms to Ada any different to the application to any other (similar) programming language?

10. Or conversely, if there exists a formalism for Ada, can it applied to other programming systems?

11. Does the Hoare [HOAR1], [HOAR2] or Bornat [BORNA] approach to correct program development have an application beyond "toy" programs?

12. Can formalisms be merely added to programming guidelines?

    a. Can a formalized approach exist within a single element of a life cycle without flowing outward (or inward)?

formed the core of the background studies for this report, are included in the appendices. The studies of the application of formalisms in verification activities is presented in the next two sections:

1. The Application of Formalisms to the Life Cycle
2. Phasing Formalisms into the Life Cycle

The responses to the issues raised earlier in this section are then answered in the Conclusions section, followed by the Recommendations for further work in this area, that concludes the report.

This work did not include an appraisal of the variety of life cycles which might form the basis of a formalized development model. As will be shown in the next section, emphasis is placed on SDS 2167 as the the vehicle through which to introduce formalisms into the software development process.

2.0 THE APPLICATION OF FORMALISMS TO THE LIFE CYCLE

An examination of the various life cycles [BALZE], [BLUM], [BOEHM], [GIDDI], [LEHMA], [ROYCE], [TULLY] reveals that each is constructed from the basic building block that is shown in Figure 2-1. Life cycles vary mainly in the "epicycles" that are

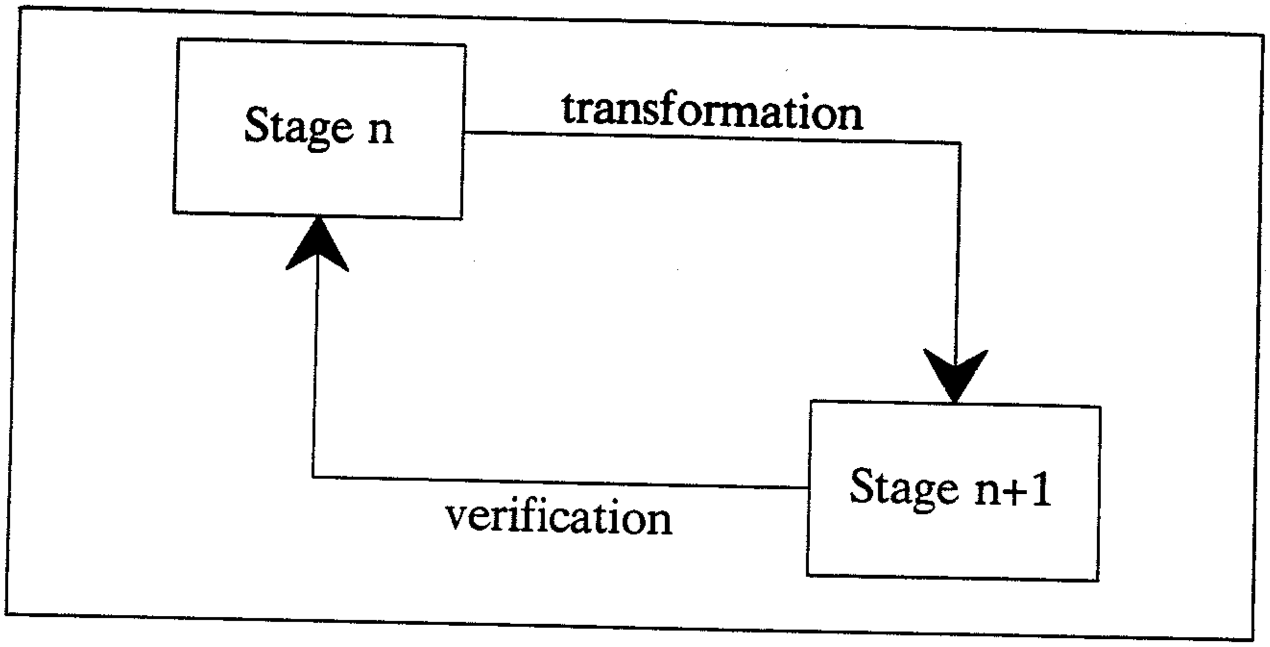


Figure 2-1: Basic Element of a Life Cycle

added to the straight line sequence of program development from requirements, through specifications (or top-level design), detailed design and implementation, to delivery and maintenance. Fundamentally, pairs of life cycle objects are joined in the forward direction by a transformational process, and in the reverse direction by a verification activity. Each stage is composed of an essential object that in turn is identified by a document, and accompanied by a number of other documents that provide plans and goals for later stages. Each of these documents epitomizes a substage in one of the sub-branches of the life cycle and is subject to same transformation and verification processes as the major stages. In essence each document is both an output from a milestone stage as well as being the input to the next or later stage. Irrespective of the epicycles or sub-branches contained in a life cycle, the basic building block is the same. Currently this basic building block of each life cycle has four elements that are candidates for the application of formalisms. These are:

- The source object, such as the set of system specifications,
- The target object, such as the code for the implementation,

- The transformation process from source to target, and
- The verification process, which confirms that the transformation process has accurately reflected the properties of the source in the target.

At the present time, the verification activities receive the greatest attention with respect to formal systems. That is, attention has been paid to the review process in that the accuracy of the target is compared to the properties suggested by the source. In the same manner as the compiler (as a transformation implementation) has replaced hand coding of machine language programs without the need for verification, so it must be expected that the transformational processes within the life cycle can be similarly replaced. The major advantage of such an application of formal processes will be the lessening of the dependence on post-transformational verification. Automatic programming has been the hope of many computer scientists since the beginning of the field to preclude such dependence.

The term has been applied to a widely varying set of technologies, but they always seem to be what is just over the horizon. As an example, in the 1950's, FORTRAN was called an automatic programming tool. In many ways it did automate the programming of arithmetic formulae. Many errors were eliminated, optimizations could be done by the translating software, and consequently higher confidence could be placed in the result. There was still considerable opportunity for error -- misused operator precedence and parentheses for example. These kinds of errors could not be caught by the compiler because these were correct programs, just not the ones intended. The attempt to answer this issue drew attention in two different directions. One was toward testing and the other was toward an even higher level specification of the program.

The purpose of testing was to demonstrate the functional performance of the software. The development of these tests required as much, if not more, analysis and development work as the program itself. It was also clear that testing could uncover only a limited number of errors. On the other hand, it is much easier to justify testing as a prelude to debugging and to quantify testing success in terms of errors located and bugs fixed. Test results could be compared with the expected results and thereby determine if the program was (partially) correct. In particular if the original problem was misunderstood, it was possible for both the programming and testing teams to do their jobs differently and incorrectly. Sometimes they would catch each other, but sometimes they would make consistent misinterpretations.

This motivated the other approach -- to move to higher level languages that would be closer to the original problem and hopefully understandable by the people with the problem to be solved. By using a higher level language, the number of interpretation (and thereby potentially error causing) steps between the problem statement and the solution specification could be reduced. This is similar to the motivation behind prototyping. The general hope is that written at the proper level, it will be possible for the original requestors to determine if it is the correct program. Then the actual program could be automatically derived through a series of transformations, program generation, and compilation steps.

Much of the work on program proving is closely connected to this approach. By putting assertions about the program into the code itself, it is hoped that the program can be automatically proved to conform to these assertions and it is also hoped that these assertions will be more directly linkable to the requirements and specifications of the original problem.

Because of the close linkage of these two approaches, it is unlikely that one will succeed without the other. Current approaches in automatic programming are along the lines of automating the generation of programs starting from very high level specifications. These very high level specifications are very much like the high level assertions used in the program proving and verification approach.

Research and development work going on now should emphasize some of the common features of the two approaches and start providing the programmer tools and interfaces that would be useful -- for example, some kind of advanced editor that would help build high level assertions and link them to the high level specifications and requirements that are supposed to be answered by the code under development. This kind of tool could be immediately useful in the context of traditional IV&V where requirements, specification and code must be linked. It will also be useful in developing code that has a high probability of fitting into any automatic verification environment developed in the future.

Some of the work on program transformations has already been applied in unstructured to structured code converters and in recursion removal optimizers. Many other transformations could be used to assist programmers. The programmer's assistant kind of work is really just a high level program building tool, but it still requires an expert programmer and and expert in the tool to use it effectively. It is doubtful whether in the very near future that a completely trusted transformational system can be built between specifications and top-level design. Thus it must be expected that those elements of transformation that cannot be formalized must be subjected to verification. This scenario also provides a paradigm for the transition from a systematic life cycle to a fully formalized system.

The following section reviews the current approaches to formal description, as they can be applied to the the objects at each stage of the life cycle.

2.1 OVERVIEW OF CURRENT APPROACHES TO FORMAL DESCRIPTION

The choice of formal description techniques for the objects of each stage of the life cycle will depend on our ability to demonstrate the interrelationships between the descriptions in successive stages. That is, a pair of formalisms that are not susceptible to either transformation or verification are not suited for use in a coordinated life cycle. Using a top down approach to program development, it is not necessary that all formal descriptive techniques have applicability throughout the whole domain of the system under development.

At the present time, software system specifications are done in the English language. While using English as a specification language has the advantage of providing easily readable, easily composed specifications there are some problems inherent with the use of English. The English language often contains inconsistencies and ambiguities that inhibit exact interpretations of the specifications. The translation required from specification language to coding is so broad due to the vast difference in media as to create transitional errors in all but the most detailed, trivial or exhaustively used system. The use of a formal language, such as Ada, to describe such specifications would greatly reduce the number of translational errors and the effort required in coding numbers. This could be used as an important first step in developing trusted verifiable software.

The use of Ada as a specification language enables the specification to be read and interpreted by a compiler-like consistency checker that is able to enforce internal consistency within the semantics. Taken to a higher level, the consistency checker may be used to check the consistency between different levels of specification. In this manner the integrity of the initial specification may be checked, level by level, down to the actual code. If the compiler being used is a trusted compiler then the consistency of all specification level transitions have been verified from the most abstract level to the actual code.

2.1.1 A COORDINATED, COMPREHENSIVE FORMALISM FOR PROGRAM DEVELOPMENT

Six differing forms of descriptive systems have been developed in the past 20 years, each of which has individualistic capabilities and applicabilities. These six (Petri Nets, temporal logic, algebraic semantics, denotational semantics, axiomatic semantics, and operational semantics) are described below, and in greater detail in Appendix A.

Petri nets can be used to specify both functionality (correctness) and performance (time constraints) of computer systems. They have a simple graphical representation so that they are comparatively easy to understand, to learn and to use, and are executable so that the simulation and prototyping of a target system can be undertaken. Since Petri nets are abstractions they can be used in a stepwise refinement methodology. However to avoid producing very large, unstructured descriptions of computer systems, it is necessary to analyze the nets to identify certain undesirable properties. Petri nets should not be used at a very low level.

Temporal logic has been widely used in specifying and verifying many program properties such as safety (partial correctness, clean behavior, mutual exclusion, deadlock freedom) and liveness (total correctness, accessibility, fairness). It has been especially useful in specifying and verifying concurrent programs, distributed systems and network protocols. Temporal logic is abstract, precise and machine processable, but lacks compositionality and is too detailed for some applications. Higher level temporal operators of interval logic are needed.

The algebraic approach is well suited to specifying and verifying abstract data types. It is mathematically well-defined so that formal analysis and proof can be performed on descriptions of program modules. Various transformations can be made automatically according to algebraic laws and equations in the same specification framework. Since the approach is abstract and modular, information hiding can be achieved, and compositionality is possible. The algebraic approach is particularly useful in describing the interfaces between modules. It is machine processable but the approach is not well suited to defining control structures and concurrency. Neither is it appropriate to be used at very high level.

Denotational semantic formalisms are well suited to specifying and verifying sequential entities in programming systems. They are especially useful in describing functional relations, and is mathematically precise so that formal proofs and transformations can be performed. Like the algebraic approach, denotational formalisms are abstract and compositional so that stepwise refinement and modularity can be achieved. They are difficult to use in connection with with concurrency; several attempts have been made to use denotational approaches to this problem but have resulted in complicated descriptions that are difficult to understand. Thus, it is not appropriate to be used at very high level.

The axiomatic model is well suited for specifying and verifying sequential program properties. It is abstract and easy to understand. The principle is so simple that it can be used directly by programmers at very low level; on the other hand the ability to identify appropriate, complete and consistent axioms is difficult! When a proof system is constructed, correctness verification can be performed automatically. On the other hand, it is not appropriate for describing concurrency, and it is very hard to find invariants for complicated structures. Thus, it is better to be used locally at low level. The majority of currently developed formal verification systems utilize this methodology.

Operational semantics is one of the earliest approaches to formal description that had practical application. This is perhaps because it is similar to a programming language itself and thus more likely to be acceptable to programmers. The operational approach is best suited to describing the functionality of an implementation. It is comparatively easy to learn and to use. It has been widely applied in many software specifications and is the basis of the only existing formal programming language standard. While it is machine processable, it is low level and subject to the same problems of verification as the implementations it is intended to describe! The style of an operational description may be more acceptable to programmers but it can only be used in low level design and must be supplemented by other verification techniques.

Based upon the above analysis and comparison, a software development model is presented in Figure 2–2. It utilizes all the above mentioned formal specification techniques and elaborates the waterfall model.

At very high levels, Petri nets are used to abstract and formalize customers informal requirements in a graphical manner. Validation, prototyping and simulation can be performed at this stage. Petri nets should be combined with temporal logic formulae for verification purposes. At the succeeding level of detailing, Petri nets can be further refined so that the control structure of the whole system can be constructed and the interfaces between various modules can be specified. Within each major component of the Petri net, modules can be designed by using the algebraic approach for data abstraction and the denotational approach for functional abstraction. Various transformations, rewriting and correctness proving can be carried out formally as this detailing is continued. At the lowest level, modules can be further refined using the operational approach and axiomatic formalisms to replace higher level algebraic and denotational definitions. Initially these transformations will be required to be formally

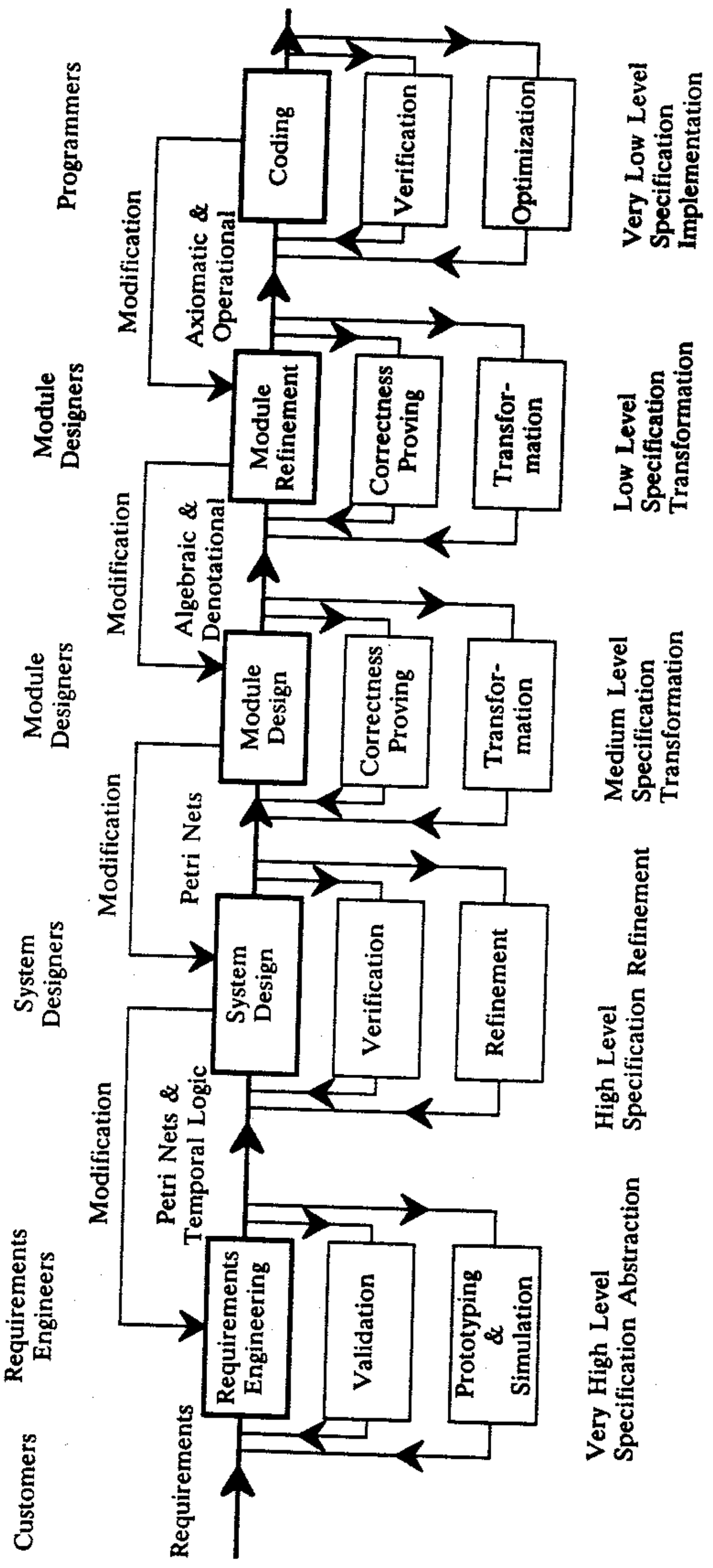


Figure 2-2: The Software Development Model

verified, but as the transformational process is itself formalized, verification will be less and less important.

The ability to interface the various specification techniques and the design of automated tools (structural editor, term rewriting system and transformer, theorem prover) remain to be investigated. Results and experiences from the CIP, VDM, and Balzer's projects will be useful in constructing this coordinated approach.

Software quality assurance can at least be partially carried out during the whole software development process in this model. Correctness has the first priority; correctness can be validated, verified and proved during each stage down to the implementation level. Performance criteria can also be simulated by Petri nets and may be verified throughout the whole development process. Comprehensibility can be better achieved by the high level structural module and interface design. It is still a open research area as to how well formalisms can guarantee software quality.

2.1.2 IMPLEMENTATION AND RUN-TIME CONSIDERATIONS

[PREST] analyzed each individual component of the Ada programming language and provided some clues as to the difficulties that have been created for formal verification by the complexity and the language. This approach was based on "traditional" axiomatic verification, though it is doubtful whether the conclusions would be too different for other techniques. As is suggested in appendix A, the axiomatic approach is probably best suited to this element of the life cycle. It is possible that a number of Ada programming language constructs will not be amenable to formal verification. This is a problem that is similar to that faced by the software engineering community in the late 1960's when structured programming was being introduced. The cultural issues of using unstructured code were eventually outweighed by the "sensibility" of the Bohm-Jacopini constructs [BOHM] on the basis of cleanliness rather than verifiability!

The Anna specification language developed at Stanford University seems to have its greatest applicability in the area of validation of the run-time behavior of Ada programs. Such technology is not as "high tech" as formal verification, but has significantly greater potential for use and application by ordinary programmers unskilled in the formal methods, and can be applied using today's technologies and tools, while the formal verification process (and tools) remain in the future.

2.1.3 CONCURRENCY, EXCEPTIONS, AND FORMALISMS

In the section above we have briefly provided an overview of the applicability of formalisms to the concurrent aspects of program development and implementation.

Far and away, concurrency is one of the most hazardous obstacles in the way of applying formal verification technology to the Ada language. By restricting the use of tasking, it appears that concurrency in Ada can be made included in the constructs to which verification can be applied. Some of the techniques employed in Communicating Sequential Processors (CSP) [BARRI] appears to be a promising possibility while other research indicates that restriction of communication to only buffers (a la Gypsy) [YOUNG] to only scalars [ODYSS], or to only those entry points in that pre-condition and postcondition assertions have been specified [TRIPA] would alleviate many of the inherent difficulties in applying formal verification technology to the Ada concurrency problems.

Several researchers [ODYSS] and [PNEUL] have recommended that access pointers to tasks not be allowed. They have also indicated that in the absence of dynamic creation of tasks, proof rules can be obtained for tasking. However, it is unlikely that this limitation will be readily accepted, particularly in the systems programming arena.

The approach used by Owicki and Gries [OWICK] to verify communicating sequential processes, Hoare's language framework for concurrent programming, is readily adapted to verification of Ada tasks. This approach consists of two distinct steps: internal verification and external verification. Internal verification consists of proving that the task is an isolated, sequential program. External verification consists of proving that, with the exception of entries, tasks do not affect any subprograms, tasks, or variables declared outside of the task being verified. External verification also requires proof that the task in question is not affected by any subprograms, tasks, or variables declared outside of the task. Again, entries are the exception to this rule. External verification is performed in two states:

1. I/O assertions on entries are made and shared variables are restricted; and

2. A proof against deadlocks and starvation is made.

Deadlock and starvation avoidance proofs are prevalent throughout parallel processing literature.

The verification of tasks also assumes the following:

1. All constructs terminate normally;
2. Subprogram calls have no side effects;
3. Assignments have no side effects; and
4. Tasks may not be assailed.

The major difficulty with exceptions [TRIPA] in the Ada language from the point of view of verification is the dynamic manner in which exceptions are propagated, and the resulting complexity that derives from attempting analysis during symbolic execution of programs in the verification step. This complexity is furthered by the fact that exceptions are propagated "as is," that could cause an unhandled exception to propagate from several levels down to a routine that has no understanding of the meaning of the exception. For example, a stack package with a private implementation that raises INDEX_ERROR in the environment of the calling procedure would be totally unexpected and either unhandled or mishandled.

Through adequate containment of the exceptions -- conversion of unhandled exceptions to some ROUTINE_ERROR on exit from a block (within a package or not), or explicit use of "OTHERS" clauses at all possible junctions (not a convenient approach) -- the complexity should be reduced.

Another matter of concern of exceptions is due to the non-specificity of the language with respect to modes of parameter passing. If a compiler passes an IN OUT variable by copy on entry and on exit, the variable may never be updated if the routine raises an exception, whereas if the variable is passed by reference, changes to the local variable may actually change the passed variable, and the value will have been updated in the presence of a raised exception.

2.2 PHASING FORMALISMS INTO THE LIFE CYCLE

One of the earliest pioneers of the introduction of formalisms into program description states that "... certain arguments given for using particular formal modes of expression in developing and proving programs correct are invalid. Emphasis on formalization is shown to have harmful effects on program development, such as the neglect of informal precision and simple formalizations."[2] A study undertaken [GEHAN] deals with the introduction of formalisms at one specific level with some moderate degree of success,

[2] Naur, Peter: "Formalization in Program Development", BIT, Vol. 22, 1982, pp. 437–453.

be acceptable in a well-established life cycle environment; a progressive, well-planned introduction of formalisms is necessary, irrespective of its immediate usefulness.

Their solution consists of introducing "trust domains" into the system under development to demonstrate that the system and its components satisfy a well-defined set of constraints. A specific language of constraint description is provided that operates over the domain of the communications between components. Though not explicitly suggested it would appear that this methodology would interface very well with an upper level description of structure and communications in the system, and with lower level assertive descriptors in the components.

As discussed previously, there are two philosophies that should be observed in the transitional process from the systematic life cycle to the full formal approach:

1. The formalisms of life cycle object description should not replace the systematic elements, but instead should work alongside "manual" methods, and

2. The progression of introduction of formal methods can be accomplished in a stepwise manner.

Five uses of formalisms in the life cycle can be identified that would lead to their later fuller use:

1. The use of formal descriptions of programming languages in the place of reference manuals;

2. The use of assertions in program modules, and insert assertions (with appropriate cross checking) in the configuration management support for program development;

3. The formalization of the contents of documentation wherever possible;

4. The use of software metrics wherever possible to provide an orthogonal set of measures of the quality of the software under configuration management; and

5. The development of the necessary tools to support these concepts.

The ultimate goal is to supplement each life cycle stage with a formalism that will appropriately "capture" the essential elements of that stage that should be tracked throughout the design process. This process will be effective if carried out in a orderly and moreover in a "real" environment. The conclusions state, however, "... formal specifications of large systems is somewhat impractical at the moment, [but] experience in using formal specifications can lead to [a] better informal specifications. The use of algebraic specifications requires practice and experience." This is a somewhat pessimistic view, one that we believe can be overcome by the use of a multiplicity of formalisms at different levels of detail. Just as the problems of program development have been materially assisted by the use of the top-down approach, with structured development, taking every advantage of the algorithmic method of "divide and conquer" (aka successive refinement), so such approaches can be usefully employed in the introduction of formalisms. It should *not* be assumed that the whole life cycle can be supported by one formal descriptive technique, *nor* should it be assumed that the whole system under development is to be described in a single object! Programs can be described adequately in pieces (modules) and be shown to be consistent (interfaceable) with upper levels of communication and structure.

We are aware of several other efforts to formalize the program development methodology, but only two that specifically took the objective of using formalisms *within* the life cycle. [NEELY] deals with the rigorous integration of "sources of assurance" while [LOGIC] restricted themselves to rigorous [but] informal verification techniques. The latter proposed inserting an assertive strategy between the constraints of a specification and the software implementation. The example given is somewhat small and little advanced from the style of [HOAR1], [HOAR2], [BORNA]. On the other hand the recommendations for further work include:

1. A system development methodology for an entire system life cycle.
2. A guide to the development and use of assertions.
3. A detailed example of rigorous, informal verification in practice.
4. Use of complexity analyzers for path analysis.

While [LOGICO] does not emphasize the use of software metrics, the last recommendation suggests the use of metrics as an ancillary assurance tool.

[NEELY] states that "a development process incorporating [formal] verification is an all-or-nothing proposition. If one tried to apply a "little dose" of the techniques then there is little to be gained. In fact, it may be money not well spent. Once a formal verification process has been applied to a system, major portions of the system are off limits to any further changes." It is doubtful whether this all-or-nothing approach would manner and with good reason. There will be no point in merely supplementing the stages with formalisms if there do not exist means to aid in the forwarding of that information into the next stage, to assist in the "fleshing out" of that information in the next stage, and to verify that the enhanced data is in conformance with the requirements of the previous stage. A large portion of the tracking of requirements can be accomplished through the use of a more sophisticated configuration management system and the ability to demonstrate the consistency of derived specifications and implementations.

It will be necessary to add the first pair of formalisms in adjacent life cycle stages, and to provide the tools to relate the properties of the objects in each stage. If we regard each of the processes of development as a transformation between objects, then we expect those objects to be expressed in a formalism. This is summarized in Figure 2-3.

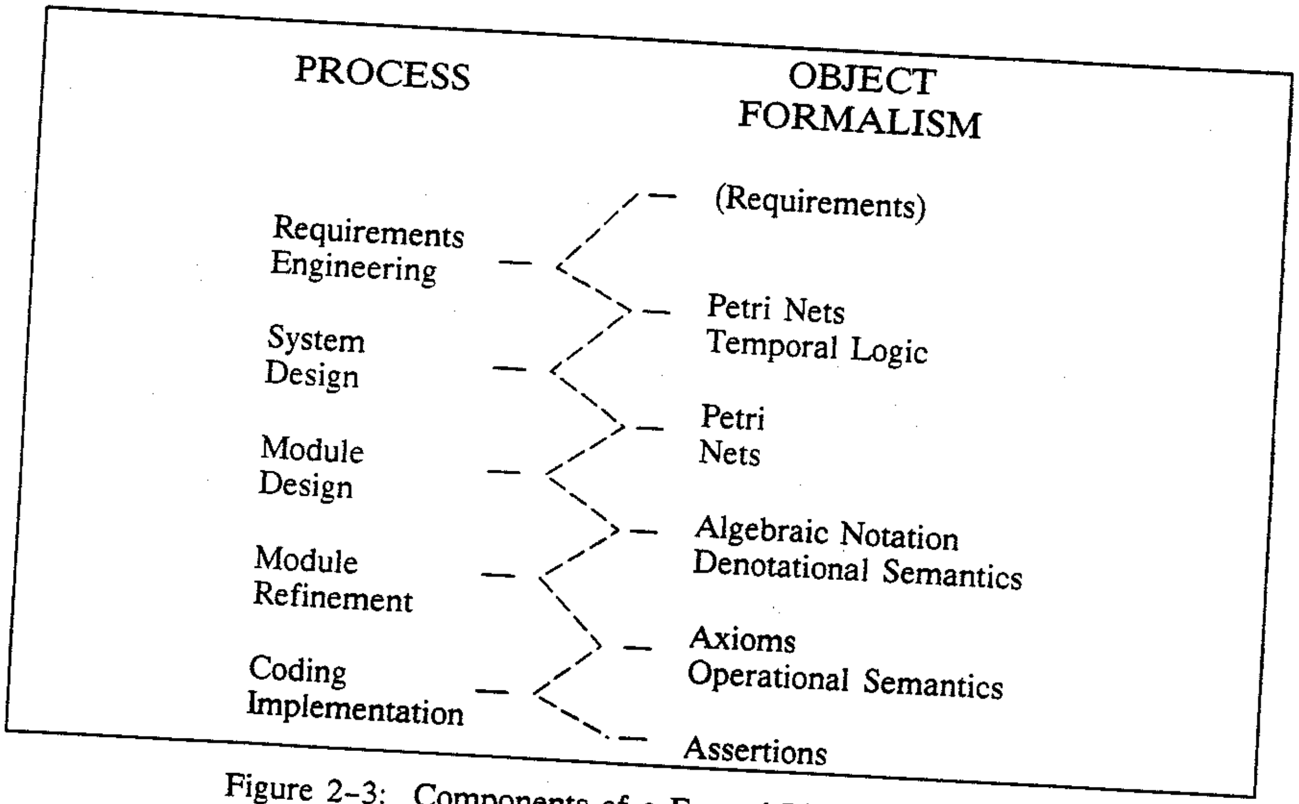


Figure 2-3: Components of a Formal Life Cycle Model

Perhaps the easiest place to start will be at the coding process, by surrounding it with the formal axioms of the design and the assertions in the run-time implementation. The axioms derived from the module design should be checked by the implementation, and can later be verified in terms of a compilation of the semantics of the module from a description of the language used. Thus, in the case of Ada, a formal description of the module should be derived from the definition of the language, and the assertions be shown to:

1. Be consistent with the compiled description, and
2. Be valid transformations of the axioms.

Working backward through the life cycle in this manner would appear to us to be more likely to succeed on the basis of demonstrated need (though admittedly being a bottom-up approach), that is useful at all times, than to insert the formalisms top-down. The danger in this approach is that (for example) the axioms derived from the design have not been verified against the requirements from the previous stage until all previous stages have been converted to a formal description. However the experience with using the formalisms will have immediate effects in raising the level of discipline needed at that particular stage and, thus, raising the quality assurance of that stage.

The classes of tools needed for support of this development environment will include:

- Formal description "compilers" -- tools that given a description of an object (or the properties of an object) in terms of a specific language, construct a formal description in generic terms. For example, given a description of (say) Ada in VDM, a compiler would develop a description of a module in VDM.
- Cross referencing theorem provers -- tools that will verify (at least) the consistency of the successive descriptions of the life cycle objects in differing formalisms.
- An enhanced configuration management system -- that will ensure the consistency of interfaces between the various elements of the program. This will include interfaces between successive objects in the life cycle (for example between the design specifications and an implementation), and between components of the implementation (driver-stub relationships).
- Symbolic executors -- that will demonstrate the correctness of implementations at various levels, thereby providing a rapid prototyping capability as well as providing a further tool for the verification of assertions -- even to the point of deriving assertions and invariants for comparison to requirements.

Consideration must be given to the impact of the introduction of formalisms in the development process. Will it be necessary, for example to modify the existing development model in order to accommodate formalisms or can the existing system exist side-by-side with the formal approach?

2.2.1 FORMAL ASSISTANTS

The Stanford University research has focused mostly on the development of a specification language, Anna, and the development of tools to support its use in the area of run time assertion checks. The language was completed several years ago, and is being used by a number of contractors and other researchers, both in real world applications and in further research. The automated support tools themselves are written in Ada, and run on a number of different hardware architectures. Neither the language, nor the support tools have been designed specifically for use in a verification environment, although much of the effort that has been completed would provide a starting basis for such an environment.

Computational Logic, Inc. is investigating the underlying logic necessary to support an Ada verification environment. Leveraging their background with the Gypsy Verification Environment, the Boyer-Moore theorem prover, and the beginning research on the underlying logic for the Rose language (a Gypsy successor), various constructs in the Ada language are to be analyzed to determine underlying logic structure necessary to support such a verification environment. This effort is more directed at preparing the underlying formalisms, and understanding the relationship between those formalism and the programming language semantics than upon the development of any additional specification languages or automated support tools.

Odyssey Research Associates, Inc. is involved in research and development of a prototype Ada verification environment. The major thrust of the effort is the extension of Anna to support real-world programs. Anna will be extended by applying it to known examples, finding and fixing the shortcomings in each of these applications. Additional effort is planned to design and develop a prototype verification environment supporting the Anna extensions with a theorem prover and proof rules and formal semantics for that portion of the Ada language being utilized.

As these various approaches to formal verification with the Ada language are progressing, it will be necessary to apply the resulting technology in order to gain experience with them and to evaluate their feasibility for development of large scale projects. To date, applications of verification technology has been performed by small groups of people on small tasks, with rather despairing results in terms of both costs and quantity of software. This situation will be no different in application of formal verification technology to the

Ada language. The community of individuals trained in the use and application of formal verification techniques is small, and the intersection of those individuals with the limited pool of talent proficient in the Ada language continues to reduce the available labor. Existing verification projects have been small in size, because that has been the only manner in that to maintain control of the complexity of the project, and to be able to support the project with automated support tools. Advances in hardware technology will improve the utility of support tools, but can not solve the personnel problem.

Once the remaining theoretical obstacles have been overcome, it will be necessary to develop automated support tools for the transformation and verification process. A survey of some of the tools under development for the support of program development environments is given in Appendix B.

3.0 CONCLUSIONS

In an earlier section we raised a number of issues that required attention. These can now be answered.

1. *What is the objective/purpose of using formalisms?*

   It is generally believed that the introduction of discipline to the software life cycle will provide an improvement in the reliability of the resulting software. The use of formal description techniques is expected to provide the ultimate discipline and thus provide the greatest expectation of correctness.

2. *How formal is formal?*[3]

   a. At a minimum it would seem that formal can merely mean discipline and order, such as would be expected in a formalized development process, as might be derived from SDS 2167. The major component of such a formal approach would be the clear definition of milestones and review points at that go–no go decisions can be made.

   b. *Can "formal" imply procedurally formal or must it always imply mathematically formal?*

      A well defined life cycle model is a clear example of a procedurally formal system. Additional formalism can be applied layer by layer to the life cycle so as to transition to a completely mathematically formal methodology.

   c. At a maximum, formal methods implies some form of mathematical approach by that the "correctness" of the product can be assured throughout the development cycle.

3. *Can a meaningful progression from the informal (disciplined procedural approach) to the formal (mathematical) be developed?*

   Yes, through abstraction and stepwise refinement such as from informal requirements to formal requirements specification.

[3] Bjorner [BJOR5] suggests that there are three levels of formalism: formal (totally mathematical), rigorous (formal but without supporting proofs), and systematic (based on formalisms but without proofs or conjectures).

a. *Is the progression of transformation of processes continuous or discrete?*

It cannot be continuous, as some radical changes must take place in the transition from a high level (specification) language to low level (implementation) language. Even in those instances in which the same programming language is intended to be both the specification and the implementation language, much of the functional capabilities are included in comments that cannot be directly transformed into code.

b. *Is there a hierarchy or structure to progressive formalism application?*

Apparently several researchers believe this to be the case and this report substantiates this expectation.

c. *Does the same formalism exist throughout the transition from procedural to mathematical approaches?*

With the proper planning, elements of formalisms, such as the pre- and post-conditions of an axiomatic system can be used in advance of the existence of a formal proof mechanism. It would *not* be counterproductive to use different mechanisms at intermediate stages of the transitional process.

d. *If the eventual goal is a completely mathematically formal approach, can a first step towards that goal be meaningfully procedural?*

The ultimate goal must be reliable, correct software. A mathematically formal approach is only the means to achieve such a goal. The plan proposed here does not seek a new mathematically formal approach but instead proposes to use existing approaches.

e. *Would it be better to consider establishing a procedural approach that would then be overlayed with a mathematical formalism, or would the formalism replace the procedures?*

Ideally, replacement would be better, but the background of software participants are different and their perceptions of software qualities are different, thus overlaying is more appropriate. Moreover, there are some benefits to a disciplined management approach that should not be wasted -- such as intuition and common sense.

f. *In an approach that uses the overlay methodology, does the procedural component stay the same or does it change as formalism is added to the overlay?*

This is a topic that requires further study.

g. *Can certain elements of the procedural model be formalized independently of other elements? Or does the formalization have to take place in layers throughout the whole model?*

Using an abstract, modular approach, combined with information hiding, formalism can be applied in a strategic manner as the abilities of the developer community improves.

h. *Is formalism just another "tool" to be used in conjunction with other models? What other choices exist?*

Much of current research concentrates on the formal verification process; an alternative approach must be the formalization of the transformational process(es) between successive stages in the life cycle.

i. *Does* **everything** *have to be formalized?*

Not necessarily, as long as the life cycle objects can be expressed in a language that can be subjected to formalized tools.

j. *Can procedural formalisms be developed that are mutable to mathematical formalisms?*

Hopefully.

4. *Where in the life cycle can formalisms be applied?*

a. *Can mathematical formalisms be melded in a life cycle (such as MIL-STD-2167A) or is it an additional layer?*

The model proposed in this report could accommodate either of these applications. It would be expected that a layered approach would be used initially as a parallel system that would be integrated into the life cycle at a later stage. If complete verifiability and proof of correctness are to be achieved formalisms will have to be integrated into all phases of the software development life cycle. Currently, the goal of provably correct software in Ada for other than trivial examples is not realizable. Neither the methods nor the tools exist that can provide an environment where this is achievable. Since this is the ultimate goal of introducing formalisms into the software development life cycle, formalisms should be introduced at all stages. Formalisms should become a part of each stage of software development. If formalisms are kept totally separate then the value of introducing formalisms will in large measure be lost. Then proofs of correspondence between the informal and the formal would be required to reach the ultimate goal. Verification technology is at a point where some formalisms can be introduced in the first stages of the life cycle and carried through at each level. As research identifies new ways of introducing formalisms they can be incorporated into the complete life cycle.

b. *If so how?*

The CIP project and VDM approach provide some clues as the means by which the integration of formalisms into a developmental system can be accomplished.

5. *Can formalism (procedural or mathematical) be assisted, guided, and/or imposed by tools?*

Formalism can be assisted and guided by tools. Both Anna and BYRON are examples of program design languages where different levels of formalism can be introduced. Particularly, in Anna mathematical formalism introduction is facilitated by a wide range of tools. Some of these tools are available now, some are being developed. Introduction of formalisms will, because of tool availability and the "state-of-art", be a phased process. However, enough tools are available now to begin.

a. *What tools currently exist at any level of formalism?*

A number of tools, with varying levels of usefulness, are available and applicable to Ada. A summary of these is given in Appendix B. These include:

1) Programming Design Languages and Program Design Environments:

- ANNA
- BYRON
- PAMELA
- RAISE

- LARCH
- MAVEN

2) Theorem Prover Based Systems (Verification)

- AFFIRM
- GYPSY
- SDVS
- Ina Jo
- VERUS
- ABEL
- EVES
- Z
- Prolog
- HDM
- VDM
- PVS
- CPS
- Boyer-Moore

The following concepts are also to be considered:

3) Automatic Test Generators

4) Automatic Documentation Systems

5) Requirements Tracking

6) Configuration Management Tools

7) Symbolic execution systems

8) Software metrics

6. *Can existing tools:*

a. *Meet the requirements of this task?*

Potentially.

b. *Be modified to accomplish the goals of introducing formalisms?*

Possibly.

c. *Be combined in an environment to fulfill the goals that they cannot individually meet?*

   Our review of existing tools shows that there is already a tendency to create systems that cover much of the life cycle. Thus it will be difficult to combine tools due to their extensive overlap. However, some considerable attention should be paid to smaller more well targeted tools that can be integrated into a complete life cycle support system.

7. *Is the formal definition of a programming language useful?*

   a. *To language processor implementors?*

      Automatic compiler derivation and implementation guidance has been enhanced by the existence of a formal definition (VDL, VDM).

   b. *To program developers?*

      Experienced program developers can use a formal definition as a language reference manual, but this requires a large degree of previous knowledge.

   c. *To verification activities?*

      Definitely, some form of definition is a necessary condition for verification; without a formal definition, no correctness can be verified.

   d. *If so, how?*

      The experiences in VDM indicate that this is a worthwhile approach to program development and verification.

8. *Is the formalism for a programming language (such as VDM) equally appropriate to:*

   a. *The description of program written in the language?*

      Yes. "Compilers" of program descriptors have been written for several formal systems.

   b. *The specification of processes for writing programs in the languages?*

      Possibly; the most useful portion of a formal description will be the meta-description of the description itself! That is, the meta-description will provide a more precise description of the semantics of the elements of the language that will be available to the programmer.

c. *The process of writing programs in the language?*

As in the previous question — possibly. For example, VDM has developed into a software development methodology rather than a single meta-language.

9. *Is the application of formalisms to Ada any different to the application to any other (similar) programming language?*

Definitely, Ada has more features than other languages such variety of type concepts, generic, concurrency, exception/interrupt handling, separate compilation.

10. *Or conversely, if there exists a formalism for Ada, can it be applied to other programming systems?*

Yes, for the same kind of von Neumann language (conventional).

One of the major deficiencies of the Ada language with respect to verification is the lack of a concise formal definition of the semantics. The recently developed VDM description is reputed to require eight linear feet of shelf space! That is an expansion factor of two orders of magnitude over the ARM. Although the ARM may be considered to provide somewhat of an **operational** semantics, it is not sufficiently formal to be applied in the use of formal verification technology. A complete semantics of the language will be necessary to develop proof rules for the language constructs, and to complete verification condition generators and other automated support technology for performing proofs. An effort is currently under way to provide a formal definition [EEC] that will hopefully provide the mathematical foundation of language semantics. Once this foundation has been laid, attention can then be turned to the development of proof rules for the constructs in the language, that are vital for the application of formal verification technology.

11. *Does the Hoare or Bornat approach to correct program development have an application beyond "toy" programs?*

It is most likely that these approaches will be useful at a low level for the design of modules in a system, and as a basis for the training of programmers. Even though [HOAR2] discusses several different approaches to formalisms, they are all applied at the same level of detail in his single example.

12. *Can formalisms be merely added to programming guidelines?*

[PREST] has pointed out with respect to the programming language Ada that the association of parameters at subprogram call points would be the ideal location to exclude aliasing [GOOD].

Although there might be a loss of efficiency, the fact that aliasing is unnecessary and complicates application of formal verification technology [YOUNG] would seem to be sufficient reason for its elimination.

The major concern in the use of access types is the possibility of aliasing (see [ODYSS] for a lengthy discussion of the matter). One possible solution to the aliasing problem with access types, presented in [TRIPA], is to define a new operator for access types that performs component copying, rather than pointer duplication. This solution is appealing with the advent of the evaluation of the Ada language, due in the latter part of the 1980s, when changes and updates based on several years of working experience with the language will be incorporated into the language. However, restrictions on parameter passing ([ODYSS], [YOUNG]) would appear to provide the same benefit with fewer changes.

If a function performs input or output or accesses non-local variables, it is said to cause "side effects." If function subprograms are truly functional they will not include side effects. If Ada functions are restricted to exclude side effects, they can be verified similarly to Gypsy function subprograms, in that these restrictions are enforced by the language.

Shared variables are the major construct in tasking that will have to be restricted (although perhaps simulated through use of other constructs using synchronization) in order to apply formal verification technology to Ada. On this matter, there is no disagreement among the researchers ([COHEN], [GOOD], [ODYSS], [TRIPA]).

a. *Can a formalized approach exist within a single element of a life cycle without flowing outward (or inward)?*

If the integration of formalisms into the life cycle impacts the basic life cycle itself, then it is more than likely that these impacts will be reflected in the relationships between the modified stages. Thus an inwardly modified stage will have an outward effect on its neighboring stages.

b. *Can a section be added to each element of the [programming] guide that is concerned with formalisms, or is it a separate issue?*

Once the formalism has been designed, it may be possible to provide guidelines at the implementation level that will materially assist the transformation and/or verification processes. Prior to this any attempts to put formalism into the guide may be dangerous.

c. *Can the formal definition of a programming language be used to create a check list of items to be investigated when using a particular language construct?*

Possibly. This would be worthy of further investigation.

13. *If so, to what effect?*

14. *What degree of training will be needed for those who will manage a formal development process, program within this domain, perform V&V in this domain, perform maintenance activities in the domain, or accept products from the domain?*

The integration of formalisms into the life cycle may have the same impact on the need for training at the beginning level as the introduction of the life cycle itself has had. On the other hand, the degree of new training will depend on the formalisms to be used and the tools that support those formalisms. Developers in different domains should learn and use the different formalisms that are best suited for their purposes. Of course, some people may need to understand more than one formalism to communicate with other developers.

15. *What can be accomplished through formalisms that cannot be accomplished by other means?*

Highly assured reliability, correctness, and productivity.

4.0 RECOMMENDATIONS

It is patently clear that the introduction of formalisms into the life cycle must be planned. To this point in time, formalisms have been applied to almost all of the individual elements of the life cycle activity, and tools have been or are being developed to assist in certain elements of those activities. What is left (that is still a major project) is to examine the interfaces between the methodologies and tools, just the same way as we examine the interfaces in a major system. A formalized program development methodology is a large system itself and should be designed as such, using its own methods of assurance, transformation and verification. It *must* be assumed that the starting point is an acceptable life cycle model. Irrespective of the varieties and niceties of various models, it is almost a foregone conclusion that the one to be chosen will be a tailored version of MIL-STD-2167-A. This plan should include three major elements:

1. The identification of the appropriate formalisms to be used at each stage of the life cycle, the information to be transferred (and transformed) from stage to stage, and the interfacing requirements necessary to integrate the formalisms into the overall life cycle model;

2. The identification of the most appropriate software development tools and environments that will serve as the carriers for the formal components, and the potential impact of the introduction of the formalisms into the tools and environments; and

3. The production of a complete software development system.

Immediate benefits can result from this planning activity that can be introduced into a development model rapidly:

1. A detailed example of rigorous, informal verification in practice;

2. A guide to the development and use of assertions for run-time verification in a fault-tolerant system;

3. The introduction of assertions and constraints (similar to the approach in [NEELY]) into the configuration management system; and

4. Use of complexity analyzers for path analysis.

As discussed previously, there are two philosophies that should be observed in the transitional process from the systematic life cycle to the full formal approach:

1. The formalisms of life cycle object description should not replace the systematic elements, but instead should work alongside "manual" methods, and

2. The progression of introduction of formal methods can be accomplished in a stepwise manner.

Five uses of formalisms in the life cycle can be identified that would lead to their later fuller use:

1. The use of formal descriptions of programming languages in the place of reference manuals;

2. The use of assertions in program modules, and insert of assertions (with appropriate cross checking) in the configuration management support for program development;

3. The formalization of the contents of documentation wherever possible;

4. The use of software metrics wherever possible to provide an orthogonal set of measures of the quality of the software under configuration management; and

5. The development of the necessary tools to support these concepts.

5.0 ACKNOWLEDGEMENTS

The authors wish to acknowledge the contributions of several others in the preparation of materials for this report. The work reported was supported in part by the Center for Innovative Technology of the Commonwealth of Virginia, Grant No. SPC-00-002, and in part by the Software Productivity Consortium (Reston VA), under the management of Mr. Paul Cohen. Much of the early preparatory work on formal definition techniques, and the acquisition of the greater part of the bibliographies, was undertaken by He Xudong, a graduate student at Virginia Tech, supported by a grant from the Institute for Information Technology (IIT) directed by Dr. Roger Ehrich. The appendix on Software Tools was prepared by John Davis, a graduate student at Virginia Tech, under the direction of Dr. Lee.

The authors also wish to acknowledge the critical reviews of this work given by the Ada Project members at the Software Productivity Consortium, including Paul Cohen, Samuel Gregory, Peggy Skalko, Jorge Diaz-Herrera (George Mason University), and Robert Mathis (consultant). Informal conversations with John Knight also provided useful insights into aspects of the topic beyond our immediate experience.

BIBLIOGRAPHY

[ARM] Ada Programming Language (ANSI/MIL-STD-1815A), US DoD, Washington, DC, 1983. [specific references also include section numbers followed by paragraph numbers in parentheses].

[BALZE] Balzer, R., Cheatham, T.E., Green, C., "Software Technology in the 1990's: Using a New Paradigm," Computer 16, 11 (November 1983), pp. 39–45.

[BARRI] Barringer, H., Mearns, I., "Axioms and Proofs for Ada Tasks," IEEE Proceedings, 29(E) 2 (March 1982), pp. 38–48.

[BJOR1] Bjorner, D. and C.B. Jones (eds.): "The Vienna Development Method: The Meta-Language," LNCS, Vol. 61, Springer-Verlag, 1978.

[BJOR2] Bjorner, D. (ed.): "Abstract Software Specification," LNCS, Vol. 86, Springer-Verlag, 1980.

[BJOR3] Bjorner, D. and O.N. Oest (eds.): "Towards a Formal Definition of Ada," LNCS, Vol. 98, Springer-Verlag, 1980.

[BJOR4] Bjorner, D. and C.B. Jones: *Formal Specification and Software Development*, Prentice-Hall, 1982.

[BJOR5] Bjorner, D.: "On the Use of Formal Methods in Software Development," 9th Software Engineering Conference, 1987, pp. 17–29.

[BLUM] Blum, B. I. (1982): "The Life Cycle – A Debate Over Alternate Models," ACM Software Engineering Notes 7, 4 (Oct.), pp. 18–20.

[BOEHM] Boehm, B.W. (1986): "A Spiral Model of Software Development and Enhancement," ACM Software Engineering Notes 11, 4 (Aug.), pp. 14–24.

[BOHM] Bohm, C., and Jacopini, G.: "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules," Communications of the ACM, Vol. 9, No. 5, May 1966, pp. 366–71.

[BORNA] Bornat, R.: *Programming from First Principles*, Prentice-Hall Int. Series in CS, Prentice-Hall International, Englewood Cliffs NY, 1987, pp. 538.

[CHEN] Chen, L., and A. Avizienis: "N-version programming: A Fault Tolerant Approach to Reliability to Software Operation," Digest FTCS-8, 8th Annual Conference on Fault Tolerant Computing Toulouse, France, June 1978, pp. 3–9.

[COHEN] Cohen, Norman H., "Ada Axiomatic Semantics: Problems and Solutions, SofTech, Inc., One Sentry Parkway, Suite 6000, Blue Bell, PA 19422-2310, May 1986.

[DAVIS] Davis, D. W.: "A.M. Turing's Original Proposal for the Development of an Electronic Computer," National Physical Laboratory, Teddington Middlesex, U.K., April 1972.

[GEHAN] Gehani, Narain: "Specifications: Formal and Information -- A Case Study," Software -- Pract. & Exp., Vol. 12, 1982, pp. 433–44.

[EEC] European Economic Community, "Draft Formal Description of the Ada Programming Language."

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

[GOOD] Good, Donald I., et al., "An Evaluation of the Verifiability of Ada," September 1980.

[HOAR1] Hoare, C. A. R., et al: "Laws of Programming," Communications of the ACM, August 1987, Vol. 30, No. 8, pp. 672–686.

[HOAR2] Hoare, C. A. R.: "An Overview of Some Formal Methods for Program Design," IEEE Computer, September 1987, pp. 85–91.

[KNUTH] Knuth, D. E.: "von Neumann's First Computer Program," Computer Surveys, Vol. 2, No. 4, Dec. 1970.

[LEHMA] Lehman, M. M. (1980): "Programs, Life Cycles, and Laws of Software Evolution," Proceedings of the IEEE.

[LEVES] Leveson, Nancy: "Software Safety," SEI Educational Module, to be published, SEI, Pittsburg, PA, 1987.

[LOGIC] LOGICON: "Computer Security Study: Rigorous Informal Verification Techniques," LOGICON, Inc., Arlington, VA, 30 September 1986.

[MILLS] Mills, Harlan, Michael Dyer and Richard Linger: "Cleanroom Software Engineering," IEEE Software, Vol. 4, No. 5, Sept. 1987, pp. 19–25.

[NEELY] Neely, Richard B.: "Rigorous Integration of Sources of Assurance," Ford Aerospace & Communications Corp., Report CSO–TR1136, 20 June 1986.

[ODYSS] Odyssey Research Associates, Inc.: "A Verifiable Subset of Ada," (Revised Preliminary Report), Odyssey Research Associates, Inc., 713 Clifton St., Ithaca, NY 14850, September 14, 1984.

[OWICK] Owicki, S., and David Gries: "Verifying Properties of Parallel Programs -- An Axiomatic Approach," Communications of the ACM, Vol. 19, No. 5, August 1976, pp. 279–86.

[PFLEE] Pfleeger, S. L.: *Software Engineering*, MacMillan Pub. Co., 1987, pp. 443.

[PNEULI] Pneuli, A., and W. P. deRoever: "Rendezvous with Ada – A Proof Theoretical View," Proceedings of the Ada TEC Conference on Ada, Arlington, VA, pp. 129–137, October 1982.

[PREST] Preston, David, Karl A. Nyberg and Robert F. Mathis: "An Investigation into the Compatability of Ada and Formal Verification Technology," to appear in the 6th National Conference on Ada Technology, March 1988.

[ROYCE] Royce, W. W. (1970): "Managing the Development of Large Software Systems: Concepts and Techniques," Proceedings of the 1970 WESCON (Los Angeles, California, August 25–28). Western Periodicals Co., North Hollywood, California, pp. A/1/1-9.

[TRIPA] Tripathi, Anand R., William D. Young and Donald I. Good: "A Preliminary Evaluation of Verifiability in Ada," Proceedings of the ACM National Conference, Nashville, TN, October 1980.

[TULLY] Tully, C. J. (1984): "System Development Models," In Proceedings of Software Process Workshop (Egham, Surrey, U.K., February 6–8). IEEE CS, Los Angeles, California, IEEE Catalog Number 84CH2044-6, pp. 37–46.

[YOUNG] Young, William D., and Donald I. Good : "Generics and Verification in Ada," Proceedings of the ACM Symposium on the Ada Language, Boston, MA, pp. 123–127, 9–11 December 1980.

APPENDIX A: SURVEY OF FORMAL DEFINITION METHODS

A.1 SOFTWARE LIFE CYCLE

The Software life cycle is the description of the *process* of software development from user requirements through design and implementation to delivery and maintenance. There are several software life cycle models [11], [29], [174] but the most popular one is the *waterfall* model which captures the main phases during a software development. The waterfall model is presented in Figure A-1, though many variants can be found.

A.1.1 SOFTWARE QUALITY

Software quality is one of the the measures of the *product* of software development. There are many criteria by which to measure the software product [142], which can be divided into three categories:

- functionality,
- performance, and
- comprehensibility.

Functionality includes *correctness, reliability, robustness*. Performance concerns *efficiency, cost–effectiveness, constraint satisfaction*, and *reusability*. Comprehensibility refers to *maintainability, understandability* and *usability*. These quality criteria are not necessarily maintained in any intermediate product, but must be assured by the intermediate phases.

A.1.2 SOFTWARE DEVELOPMENT PARTICIPANTS

Software development participants can be divided into different groups according to the phases of software life cycle: *customers* (those making requirements), *requirements engineers* (those formalizing requirements), *system designers* (those designing system structure and interfaces), *module designers* (those designing modules), *programmers* (people implementing algorithms and data structures), *maintainers* (those installing and maintaining the software product), and *users* (those using the software product, who also may be customers). People belonging to different groups may have different perceptions of the software process and software product. Usually, those at phase *n* are only responsible for the quality of intermediate product for phase *n+1* , and concern about the quality of intermediate product from phase *n–1*.

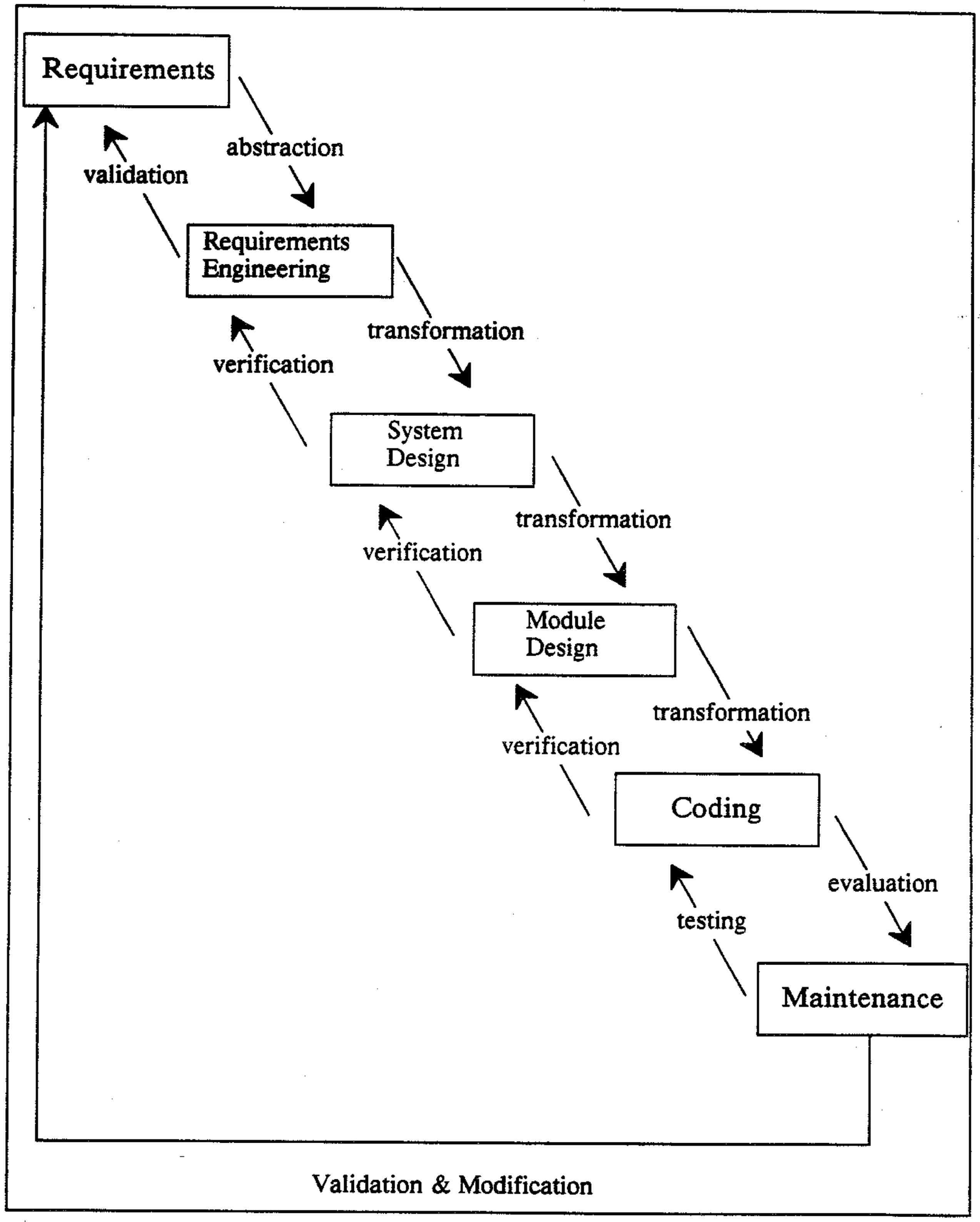


Figure A-1: Waterfall Model

A.2 ANALYSIS OF SPECIFICATION TECHNIQUES

A.2.1 REQUIREMENTS FOR SPECIFICATION TECHNIQUES

Specification as a *process*, is the activity undertaken during the software development process to specify software properties. It runs through several phases: requirements engineering, system design, and module design.

Specification as a *product* is the precise description of the result of each of the above phases. According to [77], we have *requirement specification*, *system specification*, and *module specification* in chronological order. Specification can also be classified as *functional specification* which describes the effects of a piece of software on its external environment and *performance specification* which describes the constraints on the speed and resources utilization of a piece of software [93]. Regardless whether specification refers to a process or a product, it has to be written in some kind language (notation). We can identify several requirements for specification techniques (languages) [10], [14], [141], which in turn can be divided into:

- foundation -- formal, precise;
- applicability -- abstract, expressive, modular and compositional, parallel/sequential, adaptable, machine processable, verifiable; and
- comprehensibility -- understandable, learnable, and usable.

A.2.2 INTRODUCTION TO SPECIFICATION TECHNIQUES

In the following sections, various specification techniques will be briefly introduced, more additional tutorials are given separately.

A *Petri Net* can be defined as a triple (P, T, A) where P is a set of *places* which are usually represented by circles, T is a set of *transitions* which are usually represented by bars, and A is a set of *arcs* which connect places to transitions and transitions to places. A Petri Net is often *marked* with dots at various places. The execution of a marked Petri Net is by *firing* a sequence of *enabled* transitions. A timed Petri net is one in which time information is associated with each transition. The time information specifies the minimum time units which have to elapse before the associated transition can fire and the maximum time units before which the associated transition must fire. In this approach, the meaning of a program is modeled by the behaviors of Petri Nets [130], [149].

Temporal logic is a kind of *modal logic* which deals with time concepts. It is first order logic plus some temporal operators such as *always* (represented by a square), *sometimes* (represented by a diamond), *next* (represented by a circle) and *until* (represented by a U). In temporal logic, the meaning of a program is specified by temporal formulas over the state transitions of computation sequences. The most popular temporal logic is linear time temporal logic. Others are:

- branching time temporal logic and
- interval temporal logic

[40], [100], [112], [133], [145], [152].

Algebraic approach is based on Birkhoff's *heterogeneous algebra* and was first developed by Guttag, Zilles and ADJ group [56], [64], [93]. In algebraic approach, every (abstract) data type is treated as an algebra which consists of a set of values (domains and range), a set of operations upon the domains and equations relating various operations. The meanings of entities are given implicitly via the various relationships among the operations [46], [121].

The *Denotational approach*, previously named *mathematical semantics*, is based on Lambda Calculus and Scott's *domain theory*. In the denotational approach, entities in software are mapped onto mathematical objects (list, tuple, set, function, predicate etc.) In such a definition, three parts are identified [59], [108], [158], [160]:

1. syntactic domains,
2. semantic domains and
3. semantic functions which map objects in syntactic domains to those in semantic domains.

The *Axiomatic approach* is based on the principles of Hoare logic [73] which has a set of axioms and inference rules. The meaning of a construct (expression, statement etc.) is defined by the *pre-conditions* which must be satisfied initially and *post-conditions* which must be satisfied after the evaluation or execution of the construct. One of the best known formalisms of this approach is the *predicate transformer* that was first developed by Dijkstra and later augmented by Gries [43], [62].

The *Operational approach* is the earliest formalism developed in specifying software. It is based on the theory of computation process (state transitions). The meanings of programs are defined in terms of state transitions upon an *abstract machine* which interprets the programs. The best known formalism of this approach is VDL [89], [168].

A.2.3 APPLICABILITY OF SPECIFICATION TECHNIQUES

Petri Nets can be used to specify both *functionality* (correctness) and *performance* (time constraints) of computer systems [2], [31], [72], [78], [117], [143], [148]. Having a simple graphical representation, Petri Nets are easier to understand, to learn and to use. They are executable so that simulation and prototyping can be performed [131]. As they are abstract Petri Nets can be used in a stepwise refinement methodology. [96]. To avoid producing very large, unstructured descriptions of computer systems and to avoid large computations to analyze the nets for some properties, Petri Nets should not be used at very low level. Some modification of representation of Petri Nets is needed to achieve more concise representation and to provide more information [14].

Temporal logic has been widely used to specify and verify program properties such as *safety* (partial correctness, clean behavior, mutual exclusion, deadlock freedom) and *liveness* (total correctness, accessibility, fairness) [13]. It is especially useful in specifying and verifying concurrent programs, distributed systems and network protocols [65], [66], [87], [98], [99], [100], [135]. It is abstract, precise and machine processable [118], but lacks compositionality and is useful at too low a level [14], [144]. Higher level temporal operators of interval logic are needed [88], [104], [112], [136], [152].

The *Algebraic approach* is well suited for specifying and verifying abstract data types [56], [64], [115]. Since it is mathematically well-defined, formal analysis can be performed and proofs or correctness can be constructed. Various transformations can be made automatically according to algebraic laws and equations in the same specification framework [114]. Abstraction and modularity provide a mean by which information hiding can be achieved [47], [51], [94], and the method is very useful for describing the interfaces between modules. It is machine processable [90], [111], but is not well suited for defining control structures and concurrency. Similarly it is not appropriate for use at a very high level of development.

The *Denotational approach* is well suited of specifying and verifying sequential entities [18], [23], [59], and especially in describing functional relations and recursive functions. It is precise and mathematical so that formal proof and transformation can be performed [106], [137], [158]. Like other methods it is abstract and compositional so that stepwise refinement and modular design can be achieved [26], [160]. However many attempts to deal with concurrency have resulted in complicated descriptions that are difficult to understand [8], [21], [33], [83]. Thus, the method is not appropriate to be used at a very high developmental level.

The *Axiomatic approach* is well suited for specifying and verifying the properties of sequential programs [3], [4], [73]. The method is abstract and easy to understand — well meeting the principle of least astonishment. The principle is as simple as to make it suited for use by programmers at a comparatively low developmental level. When a proof system is constructed, correctness proving can be performed automatically [60], [105], [106]. However it is not appropriate for describing concurrency [14], [32] and it is very hard to find invariants for complicated structures — post facto. Consequently its use is best at low developmental levels.

The *Operational approach* is very useful to derive the description of an implementation [103]. Since the style of description is very much like that of a programming language, it is easy to learn and to use [3], [139]. The Vienna Definition Language (VDL) has been widely applied in many software specifications [91], [168], [174], and is machine processable. Implementations have been constructed to be directly executable [175]. However the method is low level and difficult to verify. Thus, it is best used at a low level of design and supported with other verification techniques.

A.3 THE MODEL FOR SOFTWARE DEVELOPMENT

A.3.1 COMPARISON OF SPECIFICATION TECHNIQUES

Let us evaluate the various specification techniques against the requirements for specification languages provided above. Values of 1, 2 and 3 have been assigned to each methodology in terms of the degree of conformance to each requirement, with 1 implying poor, and 3 for very good. The assignments are rather subjective, being based on our own experience, biased by study of the literature. The results of study are summarized in Tables A-1, A-2 and A-3.

A.3.2 THE MODEL AND THE METHOD

Based upon the above analysis and comparison, a software development model is presented in Figure A-2. It utilizes all the above mentioned formal specification techniques and elaborates the waterfall model. At very high level in this model, Petri Nets are used to abstract and formalize customers informal requirements as part of the requirements analysis resulting in top-level specifications. Prototyping and simulation can be performed at this stage and the output with temporal logic formulas for verification purposes. At next level, the Petri Nets are further refined so that the control structure of the whole system can be constructed and the interfaces between various modules can be analyzed. At the next level, the modules can be detailed by using algebraic approach for

	Petri Nets	Temporal Logic	Algebraic Approach	Denotational Approach	Axiomatic Approach	Operational Approach
Formal	2	3	3	3	3	1
Precise	3	3	3	3	3	2

Table A-1: Foundation

	Petri Nets	Temporal Logic	Algebraic Approach	Denotational Approach	Axiomatic Approach	Operational Approach
Abstract	3	3	3	2	3	1
Expressive	3	3	4	4	4	3
Modular/ Compositional	2	1	2	3	1	3
Sequential	3	3	3	3	3	3
Parallel	3	3	1	2	2	3
Executable	3	1	1	2	1	3
Verifiable	2	3	3	3	3	1

Table A-2: Applicability

	Petri Nets	Temporal Logic	Algebraic Approach	Denotational Approach	Axiomatic Approach	Operational Approach
Understandable	3	2	1	1	3	1
Learnable	3	2	1	1	2	3
Usable	3	2	1	1	2	3

Table A-3: Comprehensibility

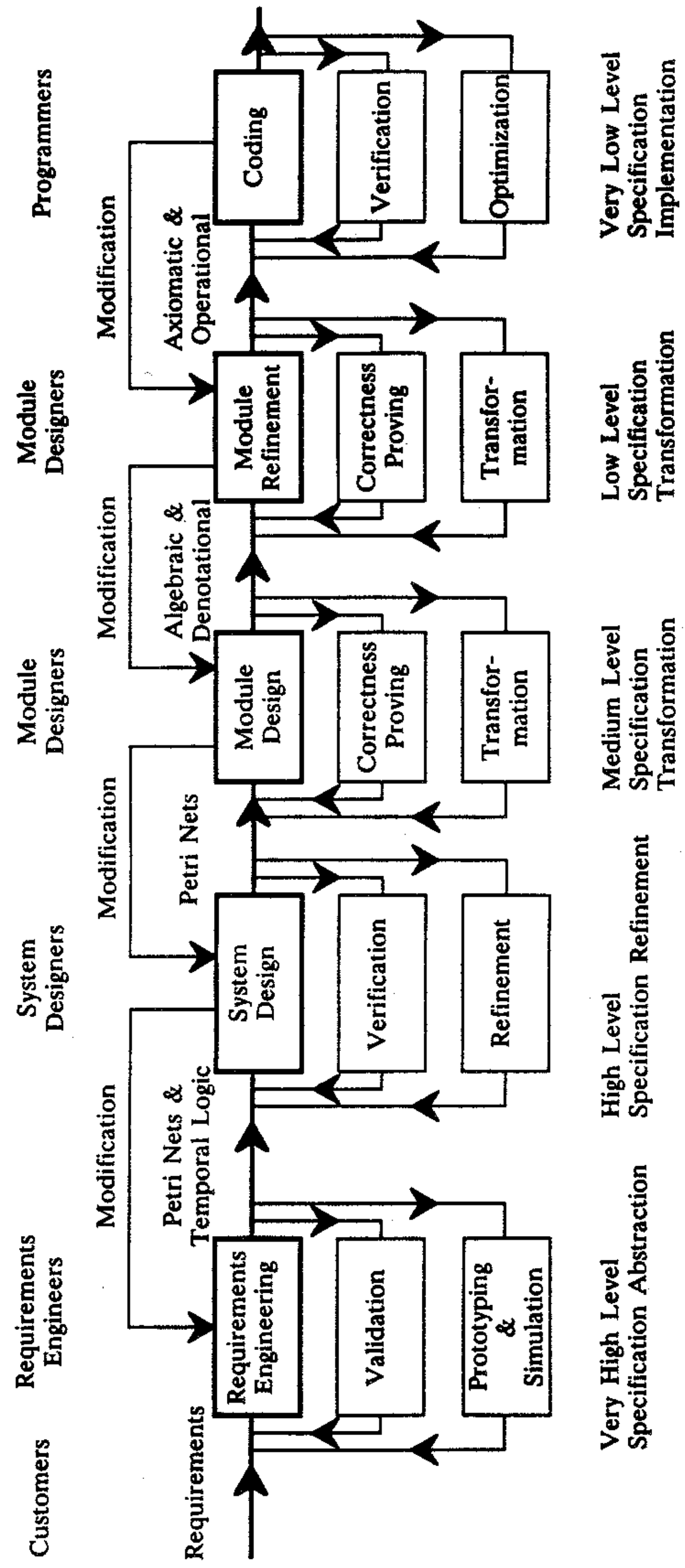


Figure A–2: The Software Development Model data abstraction and denotational approach for functional abstraction. Various transformations, term rewriting and correctness proving can be carried out. At lowest level, modules are further refined using operational and axiomatic approaches to replace upper level algebraic and denotational definitions. At each level verification can be undertaken, provided that the relationship between successive formalisms can be well defined. For example, the uppermost level of control structure must be maintained throughout the development model, and properties developed through (say) algebraic concepts be reflected in operational descriptions.

The interfacing of various specification techniques and automatic tool designing (structural editor, term rewriting system and transformer, theorem prover) are remained to be investigated. At least some results and experiences from the CIP project in Munich, [39], the VDM project, now based at the Dansk Datamatik Center [23], [26], [27], [79], [80] and Balzer's project [11] can be useful.

Formal software quality assurance can be partially carried out during the whole software development process in this model. Correctness as the first priority is apparently to be validated, verified and proved during each stage down to the implementation level. Performance criteria can also be simulated by Petri Nets and may be carried throughout the whole development process. Comprehensibility can be better achieved by the high level structural module and interface design. It is still a open research area how well formalisms can guarantee software quality.

A.4 DISCUSSION

This appendix has presented a formal software development model which employs various formal specification techniques (Petri Nets, temporal logic, and the algebraic, denotational, axiomatic and operational approaches). Though many research activities have attempted to combine various formalisms, they have dealt only with very small examples and have few implications for whole software development process. This model relates all aspects of software development: software life cycle, software quality and software participants. Research projects like CIP and VDM could have great influence on this model.

The application of formalisms in software development seems to be the right way to achieve more reliable software and to increase the software productivity as is believed and advocated by many well-known computer scientists [17], [27], [34], [44], [76], [79], [102], [107], [108], [176]. Several activities are putting these premises into practice [11], [20], [22], [27], [37], [38], [39], [63], [79], [80]. More research activities, educational efforts and practical implementations are needed to develop a feasible and practical formalistic software development methodology.

A.5 BIBLIOGRAPHY

[1] Abadi, M. and Z. Manna: "Nonclausal Temporal Deduction," Lecture Notes in Computer Science (LNCS), Vol. 193, Springer-Verlag, 1985, pp. 79-88.

[2] Agerwala, T.: "Putting Petri Nets to Work," Computer, Vol. 9, No. 3, Sept. 1979, pp. 85-94.

[3] Anderson, E. B., F. C. Belz and E. K. Blum: "Issues in the Formal Specifications of Programming Languages," Formal Description of Programming Concepts (ed. E. J. Neuhold), IFIP, North-Holland, 1978, pp. 1-30.

[4] Apt, K. R.: "Ten Years of Hoare's Logic: A Survey - Part II: Nondeterminism," Theoretical Computer Science, Vol. 28, 1984, pp. 83-109.

[5] Apt, K. R., N. Francez and W. P. de Roever: "A Proof System for Communicating Sequential Processes," ACM TOPLAS, Vol. 2, No. 3, July 1980, pp. 359-380.

[6] Auernheimer, B. and R. A. Kemmerer: "RT-ASLAN: A Specification Language for real-Time Systems," IEEE Transactions on Software Engineering, Vol. SE-12, No. 9, Sept. 1986, pp. 879-889.

[7] Back, R. J. and H. Mannila: "A Semantic Approach to Program Modularity," Inf. Control, Vol. 60 , No. 1-3, 1984, pp. 138-167.

[8] de Bakker, J. W. and J. I. Zucker: "Processes and the Denotational Semantics of Concurrency," Information and Control, Vol. 54, No. 1-2, July-Aug., 1983, pp. 70-120.

[9] de Bakker, J. W.: "Mathematical Theory of Program Correctness," Prentice-Hall, 1980.

[10] Balzer, R. and N. Goldman: "Principles of Good Software Specification and Their Implications for Specification Language," IEEE Specification of Reliable Software, 1979, pp. 58-67.

[11] Balzer, R.: "A 15 Year Perspective on Automatic Programming," IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, Nov. 1985, pp. 1257-1267.

[12] Barendregt, H. P.: "The Lambda Calculus - Its Syntax and Semantics," North-Holland, 1981.

[13] Barringer, H., R. Kuiper and A. Pnueli: "Now You May Compose Temporal Logic Specifications," Proceedings of the 16th ACM Symposium on Theory of Computing, Washington, 1984.

[14] Barringer, H.: "Formal Specification Techniques for Parallel and Distributed Systems–A Short Review," Proceedings of the 3rd Joint Ada Europe/AdaTEC Conference (ed. J. Teller), Brussels, 1984, pp. 281–294.

[15] Barringer, H., R. Kuiper and A. Pnueli: "A Compositional Temporal Approach to a CSP–Like Language," *Formal Models in Programming* (eds. E. J. Neuhold and G. Chroust), Elsevier Science Publishers, IFIP, 1985, pp. 207–227.

[16] Bartussek, W. and D. L. Parnas: "Using Assertions About Traces to Write Abstract Specifications for Software Modules," LNCS, Vol. 65, Springer–Verlag, 1978.

[17] Bauer, F. L.: "Where Does Computer Science Come From and Where Is It Going?," *Formal Models in Programming* (eds. E.J. Neuhold and G. Chroust), IFIP, North–Holland, 1985, pp. 31–44.

[18] Bekic, H.: "Programming Languages and Their Definitions," LNCS, Vol. 177, Springer–Verlag, 1984.

[19] Belkhouche, B. and J. E. Urban: "Direct Implementation of Abstract Data Types from Abstract Specification," IEEE Transactions on Software Engineering, Vol. SE–12, No. 5, May, 1986.

[20] Berg, H. K., W. E. Boebert, W. R. Franta and T. G. Moher: "Formal Methods of Program Verification and Specification," Prentice–Hall, 1982.

[21] Berry, D. M.: "A Denotational Semantics for Shared Memory Parallelism and Nondeterminism," Acta Informatica, Vol. 21, 1985, pp. 599–627.

[22] Berzins, V. and M. Gray: "Analysis and Design in MSG.84: Formalizing Functional Specification," IEEE Transactions on Software Engineering, Vol. SE–11, No. 8, Aug. 1985, pp. 657–670.

[23] Bjorner, D. and C. B. Jones (eds.): "The Vienna Development Method: The Meta–Language," LNCS, Vol. 61, Springer–Verlag, 1978.

[24] Bjorner, D. (ed.): "Abstract Software Specification," LNCS, Vol. 86, Springer–Verlag, 1980.

[25] Bjorner, D. and O. N. Oest (eds.): "Towards a Formal Definition of Ada," LNCS, Vol. 98, Springer–Verlag, 1980.

[26] Bjorner, D. and C. B. Jones: "Formal Specification and Software Development," Prentice-Hall, 1982.

[27] Bjorner, D.: "On The Use of Formal Methods in Software Development," 9th Software Engineering Conference, 1987, pp. 17-29.

[28] Blikle, A.: "A Metalanguage for Naive Denotational Semantics," Collana Cnet, Vol. 104, 1984.

[29] Boehm, B. W.: "Software Engineering," IEEE Computer, C-25, No. 12, 1976, pp. 1226-1241.

[30] Boyer, R. S. and J. S. Moore (eds.): "The Correctness Problem in Computer Science," Academic Press, 1981.

[31] Brauer, W. (ed.): "Net Theory and Applications," LNCS, Vol. 84, Springer-Verlag, 1980.

[32] Brookes, S. D.: "An Axiomatic Treatment of a Parallel Programming Language," LNCS, Vol. 193, Springer-Verlag, 1985, pp. 41-60.

[33] Broy, M.: "Denotational Semantics of Concurrent Programs with Shared Memory," 1984.

[34] Broy, M. and P. Pepper: "Program Development As A Formal Activity," IEEE Transactions on Software Engineering, Vol. SE-7, No. 1, Jan. 1981.

[35] Brum, A. D. and W. Bohm: "The Denotational Semantics of Dynamic Networks of Processes," ACM TOPLAS, Vol. 7, No. 4, Oct. 1985, pp. 656-679.

[36] Burstall, R. and B. Lampson: "A Kernel Language for Abstract Data Types and Modules," LNCS, Vol. 173, Springer-Verlag, 1984, pp. 1-50.

[37] Chen, R. S. and R. T. Yeh: "Formal Specification and Verification of Distributed System," IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, 1983.

[38] Chi, U. H.: "Formal Specification of User Interfaces: A Comparison and Evaluation of Four Axiomatic Approaches," IEEE Transactions on Software Engineering, Vol. SE-11, 8 (Aug. 1985), pp. 671-685.

[39] CIP Language Group: "The Munich Project CIP: Volume I: The Wide Spectrum Language-CIPL," LNCS, Vol. 183, Springer-Verlag, 1985.

[40] Clarke, E.M. and E.A. Emerson: "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic," LNCS, Vol. 131, Springer-Verlag, 1981, pp. 52-71.

[41] Clarke, E. M., M. C. Browne, E. A. Emerson and A. P. Sistla: "Using Temporal Logic for Automatic Verification of Finite State Systems," *Logics and Models of Concurrent System* (ed. K. R. Apt), 1985, pp. 3-26.

[42] Diaz, J. and I. Ramos (eds.): "Formalization of Programming Concepts," LNCS, Vol. 107, Springer-Verlag, 1981.

[43] Dijkstra, E. W.: "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," Communications of the ACM, Vol. 18, No. 8, Aug. 1975.

[44] Dijkstra, E. W.: "A Discipline of Programming," Prentice-Hall, 1976.

[45] Donahue, J.: "Complementary Definitions of Programming Language Semantics," LNCS, Vol. 42, Springer-Verlag, 1976.

[46] Ehrig, H. and B. Mahr: "Fundamentals of Algebraic Specification - Equations and Initial Semantics," Springer-Verlag, 1985.

[47] Ehrig, H. and H. Weber: "Algebraic Specification of Modules," *Formal Models in Programming* (eds. E.J. Neuhold and G. Chroust), North-Holland, IFIP, 1985, pp. 231-258.

[48] Fickas, S. F.: "Automating the Transformational Development of Software," IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, Nov. 1985, pp. 1268-1277.

[49] Flan, L.: "A Unified Approach to the Specification and Verification of Abstract Data Types," IEEE Specification of Reliable Software, 1979, pp. 162-169.

[50] Ganzinger, H.: "Transforming Denotational Semantics into Practical Attribute Grammars," LNCS, Vol. 94, Springer-Verlag, 1980, pp. 1-69.

[51] Ganzinger, H.: "Modular Compiler Descriptions Based on Abstract Semantic Data Types," LNCS, Vol. 154, Springer-Verlag, 1983, pp. 237-249.

[52] Gaudael, M. C.: "Compiler Generation From Formal Definition of Programming Languages, A Survey," LNCS, Vol. 107, Springer-Verlag.

[53] Gaudael, M. C.: "Specification of Compilers as Abstract Data Type Representations," LNCS, Vol. 94, Springer-Verlag, 1980, pp. 140-164.

[54] Gehani, N. and A.D. McGettrick (eds.): "Software Specification Techniques," Addison-Wesley, 1986.

[55] Goguen, J. A.: "More Thoughts on Specification and Verification," ACM SIGSOFT Software Engineering Notes, Vol. 6, No. 3, 1981.

[56] Goguen, J. A., J. W. Thatcher and E. G. Wagner: "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," *Current Trends in Programming Methodology*, Vol. IV (ed. R.T. Yeh), Prentice-Hall, 1978, pp. 80–149.

[57] Goguen, J. A. and K. P. Ghoni: "Algebraic Denotational Semantics Using Parameterized Abstract Modules," LNCS, Vol. 107, Springer-Verlag, 1981, pp. 292–309.

[58] Goldberg, A. T.: "Knowledge-Based Programming: A Survey of Program Design and Construction Techniques," IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, July 1986.

[59] Gordon, M. J. C.: "The Denotational Description of Programming Languages," Springer-Verlag, 1979.

[60] Gordon, M. J., A. J. Milner and C. P. Wadsworth: "Edinburgh LCF," LNCS, Vol. 78, Springer-Verlag, 1979.

[61] Gries, D. (ed.): "Programming Methodology," Springer-Verlag, 1978.

[62] Gries, D.: "The Science of Programming," Springer-Verlag, 1981.

[63] Guttag, J. and J. J. Horning: "Formal Specification as a Design Tool," Conference Record of the 7th Annual ACM Symposium on the Principles of Programming Languages, Las Vegas, January 28–30, 1980, pp. 251-261.

[64] Guttag, J.: "Notes on Type Abstraction," IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, Jan. 1980.

[65] Hailpern, L.: "Verifying Concurrent Processes Using Temporal Logic," Springer-Verlag, 1982.

[66] Hailpern, L.: "Tools for Verifying Network Protocols," *Logics and Models of Concurrent System* (ed. K. R. Apt), 1985, pp. 57–76.

[67] Harel, D. and A. Pnueli: "On The Development of Reactive Systems," *Logics and Models of Concurrent System* (ed. K. R. Apt), 1985, pp. 477–498.

[68] Hayes, I. J.: "Specification Directed Module Testing," IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, Jan. 1986, pp. 124–133.

[69] Heitmeyer, C. and J. D. McLean: "Abstract Requirement Specification: A New Approach and Its Application," IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, Sept. 1983.

[70] Henderson, P.: "Functional Programming, Formal Specification and Rapid Prototyping," IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, Feb. 1986, pp. 241-250.

[71] Heninger, K. L.: "Specifying Software Requirements for Computer Systems: New Techniques and Their Applications," IEEE Specification of Reliable Software, 1979, pp. 1-14.

[72] Herzag, O.: "Static Analysis of Concurrent Processes for Dynamic Properties Using Petri Nets," LNCS, Vol. 70, Springer-Verlag, 1979, pp. 66-90.

[73] Hoare, C. A. R.: "An Axiomatic Basis for Computer Programming," Communications of the ACM, Vol. 12, No. 10, 1969, pp. 576-583.

[74] Hoare, C. A. R.: "Proof of Correctness of Data Representation," Acta Informatica, Vol. 1, No. 1, 1972, pp. 271-281.

[75] Hoare, C. A. R.: "Communicating Sequential Processes," Communications of the ACM, Vol. 21, No. 8, 1978, pp. 666-677.

[76] Hoare, C. A. R. et al: "Laws of Programming," Communications of the ACM, Vol. 30, No. 8, Aug. 1987, pp. 672-687.

[77] Horning, J. J: "Program Specification: Issues and Observations," LNCS, Vol. 134, Springer-Verlag, 1982.

[78] Jensen, K., M. Kyng and O. L. Madsen: "A Petri Net Definition of a System Description Language," LNCS, Vol. 70, Springer-Verlag, 1979, pp. 348-368.

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

[80] Jones, C. B.: "Systematic Development Using the VDM Approach," Prentice-Hall, 1986.

[81] Jones, C.: "A Survey of Programming Design and Specification Techniques," IEEE Specification of Reliable Software, 1979, pp. 91-103.

[82] Jones, N. D., D. A. Schmidt: "Compiler Generation from Denotational Semantics," LNCS, Vol. 94, Springer-Verlag, 1980.

[83] Keller, R. M.: "Denotational Models for Parallel Programs with Indeterminate Operators," *Formal Description of Programming Concepts* (ed. E. J. Neuhold), 1978, pp. 337-366.

[84] Koymans, R., R. K. Shyamasundar, W.P. de Roever, R. Gerth and S. Arun-Kumar: "Compositional Semantics for Real-Time Distributed Computing," LNCS, Vol. 193, Springer-Verlag, 1985, pp. 167-189.

[85] Kroger, F.: "A Uniform Logical Basis for the Description, Specification and Verification of Programs," *Formal Description of Programming Concepts* (ed. E.J. Neuhold), IFIP, North-Holland, 1978, pp. 441-460.

[86] Lamport, L.: "TIMESETS - A New Method for Temporal Reasoning about Programs," LNCS, Vol. 131, Springer-Verlag, 1981, pp. 177-196.

[87] Lamport, L.: "Specifying Concurrent Program Modules," ACM TOPLAS, Vol. 5, No. 2, 1983, pp. 190-222.

[88] Lamport, L.: "An Axiomatic Semantics of Concurrent Programming Languages," Logics and Models of Concurrent Systems (ed. K. R. Apt), NATO ASI Series, Vol. F13, Springer-Verlag, 1985, pp. 77-122.

[89] Lee, JAN: "Computer Semantics," Van Nostrand, 1972.

[90] Leszczylowski, J. and M. Wirsing: "A System for Reasoning Within and About Algebraic Specifications," LNCS, Vol. 137, Springer-Verlag, 1982, pp. 257-282.

[91] Li, W.: "An Operational Semantics for Ada Multitasking and Exception Handling," Proceedings of AdaTEC Conference, Washington, 1982.

[92] Li, W. and P.E. Lauer: "Using the Structural Operational Approach to Express True Concurrency," *Formal Models in Programming* (eds. E.J. Neuhold and G. Chroust), Elsevier Science Publishers, IFIP, 1985, pp. 373-398.

[93] Liskov, B.H. and S.N. Zilles: "Specification Techniques for Data Abstraction," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, 1975, pp. 7-19.

[94] Liskov, B. and J. Guttag: "Abstraction and Specification in Program Development," McGraw-Hill, 1986.

[95] Loeckx, J. and K. Sieber: "The Foundation of Program Verification," John Wiley and Sons, 1984.

[96] Maiocchi, M.: "The Use of Petri Nets in Requirements and Functional Specification," *System Description Methodologies* (eds. Teichroew, D. and G. David), North-Holland, IFIP, 1983, pp. 253-272.

[97] Mandrioli, D., R. Zicari, C. Ghezzi and F. Tisato: "Modeling the Ada Task System By Petri Nets," Computer Language, Vol. 10, No. 1, 1985, pp. 43-61.

[98] Manna, Z. and A. Pnueli: "Verification of Concurrent Programs: Temporal Proof Principles," LNCS, Vol. 131, Springer-Verlag, 1981, pp. 200-252.

[99] Manna, Z. and A. Pnueli: "Verification of Concurrent Programs: the Temporal Framework," *The Correctness Problem in Computer Science* (eds. R.S. Boyer and J.S. Moore), Academic Press, 1981, pp. 215-274.

[100] Manna, Z. and A. Pnueli: "How to Cook a Temporal Proof System for Your Pet Language," 10th Annual ACM Symposium on the Principles of Programming Languages, Austin, Texas, 1983, pp. 141-154.

[101] Manna, Z. and P. Wolper: "Synthesis of Communicating Processes from Temporal Logic Specifications," ACM TOPLAS, Vol. 6, No. 1, 1984, pp. 68–93.

[102] Manna, Z.: "Mathematical Theory of Computation," McGraw–Hill, 1974.

[103] Mazaher, S. and D. M. Berry: "Deriving a Compiler from An Operational Semantics Written in VDL," Computer Language, Vol. 10, No. 2, 1985, pp. 147–164.

[104] McLean, J.: "A Complete System of Temporal Logic for Specification Schemata," LNCS, Vol. 164, Springer–Verlag, 1983.

[105] Meyer, A. R. and J. Y. Halpern: "Axiomatic Definitions of Programming Languages: a Theoretical Assessment," Conference Record of the 7th Annual ACM Symposium on the Principles of Programming Languages, MIT, LCS TM 163, Jan. 1980, pp. 203–212.

[106] Meyer, A. R. and J. Y. Halpern: "Axiomatic Definition of Programming Language II," Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages, Williamsburg, VA, Jan. 26–28, 1981, pp. 139–148.

[107] Meyer, B.: "On Formalization in Specifications," IEEE Software, Jan. 1985, pp. 6–26.

[108] Milne, R. and C. Strachey: "A Theory of Programming Language Semantics," John Wiley and Sons, 1976.

[109] Milne, R.: "Transforming Predicate Tranformers," *Formal Description of Programming Concepts* (ed. E. J. Neuhold), IFIP, North–Holland, 1978, pp. 31–66.

[110] Milner, R.: "A Calculus for Communicating Systems," LNCS, Vol. 92, Springer–Verlag, 1980.

[111] Moitra, A.: "Direct Implementation of Algebraic Specification of Abstract Data Types," IEEE Transactions on Software Engineering, Vol. SE–8, No. 1, Jan. 1982.

[112] Moszkowski, B. and Z. Manna: "Reasoning in Interval Temporal Logic," LNCS, Vol. 164, Springer–Verlag, 1983, pp. 360–370.

[113] Moszkowski, B. C.: "Executing Temporal Logic Programs," Cambridge University Press, 1986.

[114] Musser, D. R.: "Abstract Data Type Specification in the Affirm System," IEEE Specification of Reliable Software, 1979, pp. 47–57.

[115] Nakajima, R., M. Honda and H. Nakahara: "Describing and Verifying Programs with Abstract Data Types," *Formal Description of Programming Concepts* (ed. E.J. Neuhold), IFIP, North-Holland, 1978, pp. 527-556.

[116] Neuhold, E. J. and G. Chroust (eds.): "Formal Models in Programming," North-Holland, IFIP, 1985.

[117] Nelson, R. A. et al.: "Casting Petri Nets into Programs," IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, Sept. 1983, pp. 590-602.

[118] Ngnyen, V., A. Demers, D. Gries and S. Owicki: "Behavior: A Temporal Approach to Process Modeling," LNCS, Vol. 193, Springer-Verlag, 1985, pp. 237-254.

[119] Nielsen, M., G. Plotkin and G. Winskel: "Petri Nets, Event Structures and Domains," LNCS, Vol. 70, Springer-Verlag, 1979, pp. 266-284.

[120] Nielsen, C. B. et al.: "A View of Formal Semantics," Doc. No. ISO/TC97/SC22/WG10/N227, FS/N025, International Standards Organisation Tech. Comm. 97, Britsh Standards Institute, London, U.K., 1986.

[121] Nivat, M. and J. C. Reynolds (eds.): "Algebraic Methods in Semantics," Cambridge University Press, 1985.

[122] O'Donnell, M. J.: "Equational Logic as a Programming Language," MIT Press, 1983.

[123] Olderog, E. R. and C. A. R. Hoare: "Specification-Oriented Semantics for Communicating Procsses," Acta Informatica 23, pp. 9-66, 1986.

[124] Owicki, S. and D. Gries: "Verifying Properties of Parallel Programs - An Axiomatic Approach," Communications of the ACM, Vol. 19, No. 5, Aug. 1976, pp. 279-286.

[125] Owicki, S.: "An Axiomatic Proof Technique for Parallel Programs," Acta Informatica, Vol. 6, 1976, pp. 319-340.

[126] Pagan, F. G.: "Formal Specification of Programming Languages: A Panoramic Primer," Prentice-Hall, 1981.

[127] Parnas, D. L.: "A Technique for Software Module Specification with Examples," Communications of the ACM, Vol. 15, No. 5, 1972, pp. 330-336.

[128] Parnas, D. L.: "The Use of Precise Specifications in the Development of Software," Proceedings of IFIP 77, 1977, pp. 861-868.

[129] Parnas, D. L.: "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, Vol. 15, No. 12, 1972, pp. 1053–1058.

[130] Peterson, J. L.: "Petri Nets," Computing Surveys, Vol. 9, No. 3, Sept. 1977.

[131] Peterson, J. L.: "Petri Net Theory and the Modeling of Systems," Prentice-Hall, 1981.

[132] Plotkin, G. D.: "An Operational Semantics for CSP," Proceedings of IFIP Working Conference on Formal Description of Programming Concepts II, North-Holland, 1982, pp. 199–223.

[133] Pnueli, A.: "The Temporal Semantics of Concurrent Programs," Theoretical Computer Science, Vol. 13, 1981, pp. 45–60.

[134] Pnueli, A. and W. P. DeRoever: "Rendezvous with Ada – A Proof Theoretical View," Proceedings of the AdaTEC Conference, Crystal City, 1982, pp. 129–136.

[135] Pnueli, A.: "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," LNCS, Vol. 224, Springer–Verlag, 1986.

[136] Pnueli, A.: "In Transition from Global to Modular Temporal Reasoning about Programs," *Logics and Models of Concurrent System* (ed. K. R. Apt), 1985, pp. 123–146.

[137] Polak, W.: "Program Verification Based on Denotational Semantics," Conference Record of the 8th Annual ACM Symposium on the Principles of Programming Languages, pp. 149–158.

[138] Polak, W.: "Compiler Specification and Verification," LNCS, Vol. 124, Springer–Verlag, 1981.

[139] Pratt, V.R.: "Using Graphs to Understand PDL," LNCS, Vol. 131, Springer–Verlag, 1981, pp. 387–396.

[140] Queille, J. P. and J. Sifakis: "Specification and Verification of Concurrent Systems in CESAR," LNCS, Vol. 137, Springer–Verlag, 1982, pp. 337–351.

[141] Ramamoorthy, C. V. and H. H. So: "Survey of Principles and Techniques of Software Requirements and Specification," Software Engineering Techniques 2, 1977, pp. 265–318.

[142] Ramamoorthy, C. V. et al.: "Software Quality and Requirement Specification," IEEE 1986 International Conference on Computer Languages, 1986, pp. 75–83.

[143] Reisig, W.: "On the Semantics of Petri Nets," *Formal Models of Programming* (ed. J. C. Neuhold and G. Chroust), IFIP, North-Holland, 1985, pp. 347-372.

[144] de Roever, W. P.: "The Quest for Compositionality: A Survey of Assertion-Based Proof Systems for Concurrent Programs, Part I: Concurrency Based on Shared Variables," *Formal Models of Programming* (eds. E. J. Neuhold and G. Chroust), IFIP, North-Holland, 1985, pp. 157-180.

[145] Rescher, N. and A. Urquhart: "Temporal Logic," Springer-Verlag, 1971.

[146] Reynolds, J. C.: "An Introduction to Specification Logic," LNCS, Vol. 164, Springer-Verlag, 1983, pp. 371-382.

[147] Royer, V.: "Transformations of Denotational Semantics in Semantic Directed Compiler Generation," ACM SIGPLAN Notices, Vol. 21, No. 7, July 1986.

[148] Rozenberg, G. (ed.): "Advances in Petri Nets 1984," LNCS, Vol. 188, Springer-Verlag, 1984.

[149] Rozenberg, G. and P. S. Thiagarajan: "Petri Nets: Basic Notions, Structure, Behavior," LNCS, Vol. 224, Springer-Verlag, 1986, pp. 585-668.

[150] Sain, I.: "The Reasoning Powers of Burstall's (Modal Logic) and Pnueli's (Temporal Logic) Program Verification Methods," LNCS, Vol. 193, Springer-Verlag, 1985, pp. 302-319.

[151] Schwartz, R. L. and P. M. Melliar-Smith: "Temporal Logic Specification of Distributed Systems," Proceedings of the 2nd International Conference on Distributed Computing Systems, Paris, France, 1981, pp. 446-454.

[152] Schwartz, R. L., P. M. Melliar-Smith and F. H. Vogt: "An Interval-Based Temporal Logic," LNCS, Vol. 164, Springer-Verlag, 1983, pp. 443-457.

[153] Schwartz, R. L. and P. M. Melliar-Smith: "From State Machines to Temporal Logic: Specification Methods for Protocol Standards," *The Analysis of Concurrent Systems* (eds. B. Denvir et al), Springer-Verlag, 1985.

[154] Schwartz, J.: "Denotational Semantics of Parallelism," LNCS, Vol. 70, Springer-Verlag, 1979, pp. 191-202.

[155] Sethi, R.: "A Case Study in Specifying the Semantics of A Programming Language," Conference Record of the 7th Annual ACM Symposium on the Principles of Programming Languages.

[156] Simon, H. A.: "Whether Software Engineering Needs to Be Artificially Intelligent," IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, July 1986, pp. 726-732.

[157] Staunstrup, J. (ed.): "Program Specification," LNCS, Vol. 134, Springer–Verlag, 1982.

[158] Stoy, J. E.: "Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory," The MIT Press, 1977.

[159] Swartout, W. and R. Balzer: "On the Inevitable Intertwining of Specification and Implementation," Communications of the ACM, Vol. 25, No. 7, pp. 438–440.

[160] Tennent, R. D.: "The Denotational Semantics of Programming Languages," Communications of the ACM, Vol. 19, No. 8, 1976.

[161] Tennent, R. D.: "Semantical Analysis of Specification Logic," LNCS, Vol. 193, Springer–Verlag, 1985, pp. 373–386.

[162] Tiuryn, J.: "A Survey of the Logic of Effective Definitions," LNCS, Vol. 125, Springer–Verlag, 1981, pp. 198–245.

[163] Trakhtenbrot, B. A., J. Y. Halpern and A. R. Meyer: "From Denotational to Operational and Axiomatic Semantics for ALGOL–Like Languages: A Overview," LNCS, Vol. 164, Springer–Verlag, 1983, pp. 474–500.

[164] Urban, S. D., J. E. Urban and W. D. Dominick: "Utilizing an Executable Specification Language for An Information System," IEEE Transactions on Software Engineering, Vol. SE–11, No. 7, July 1985, pp. 598–605.

[165] Watt, D. A.: "Executable Semantic Description," Software – Practice and Experience, Vol. 16(1), Jan. 1986, pp. 13–43.

[166] Weber, H. and H. Ehrig: "Specification of Modular Systems," IEEE Transactions on Software Engineering, Vol. SE–12, No. 7, July 1986, pp. 784–798.

[167] Wegner, P. (ed.): "Research Directions in Software Technology," MIT Press, 1979.

[168] Wegner, P.: "The Vienna Definition Language," Computing Surveys, Vol. 4, No. 1, 1972, pp. 5–63.

[169] Wirsing, M.: "Denotational Semantics of Algebraic Specification Languages," *Formal Models in Programming* (eds. E. J. Neuhold and G. Chroust), IFIP, North–Holland, 1985, pp. 259–284.

[170] Wolper, P.: "Specification and Synthesis of Communicating Processes Using an Extended Temporal Logic," 9th Annual ACM Symposium on Principles of Programming Language, 1982, pp. 20–33.

[171] Yau, S. S. and J. J. P. Tsai: " A Survey of Software Design Techniques," IEEE Transactions on Software Engineering, Vol. SE–12, No. 6, June, 1986.

[172] Yau, S. S. and M. U. Caglayan: "Distributed Software Design Representation Using Modified Petri Nets," IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, Nov. 1983.

[173] Yeh, R. T. (ed.): "Current Trends in Programming Methodology: Vol. I," Prentice-Hall, Englewood Cliffs, NJ, 1978.

[174] Zave, P.: "An Operational Approach to requirement Specification for Embedded Systems," IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, 1982.

[175] Zave, P. and W. Schell: "Salient Features of An Executable Specification Language and Its Environment," IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, pp. 312-325.

[176] Zemanek, H.: "Formal Definition: The Hard Way," *Formal Models in Programming* (eds. E.J. Neuhold and G. Chroust), IFIP, North-Holland, 1985.

A.6 OVERVIEWS OF FORMAL DESCRIPTION TECHNIQUES

A.6.1 PETRI NETS AS A HIGH LEVEL SPECIFICATION TOOL

A.6.1.1 GENERAL CONCEPTS

Petri Nets are a graphical methodology for the description of a high level of detail of sequential and concurrent processes in which actions are "triggered" as the result of the existance of predefined conditions. A Petri Net can be defined as a triple (P,T,A) where P is a set of *places* which are usually represented by circles, T is a set of *transitions* which are usually represented by bars, and A is a set of *arcs* which connect places to transitions and transitions to places. A net can be *marked* with dots in various places, the execution of which is by the *firing* a sequence of *enabled* transitions. A *timed* Petri Net is a one in which timing information is associated with each transition. The timing information specifies the minimum number of time units which must elapse before the associated transition can fire and the maximum number of time units before which the associated transition must fire. An example that explains these concepts can be found in Figure A-3.

A.6.1.2 CHARACTERISTICS OF PETRI NETS

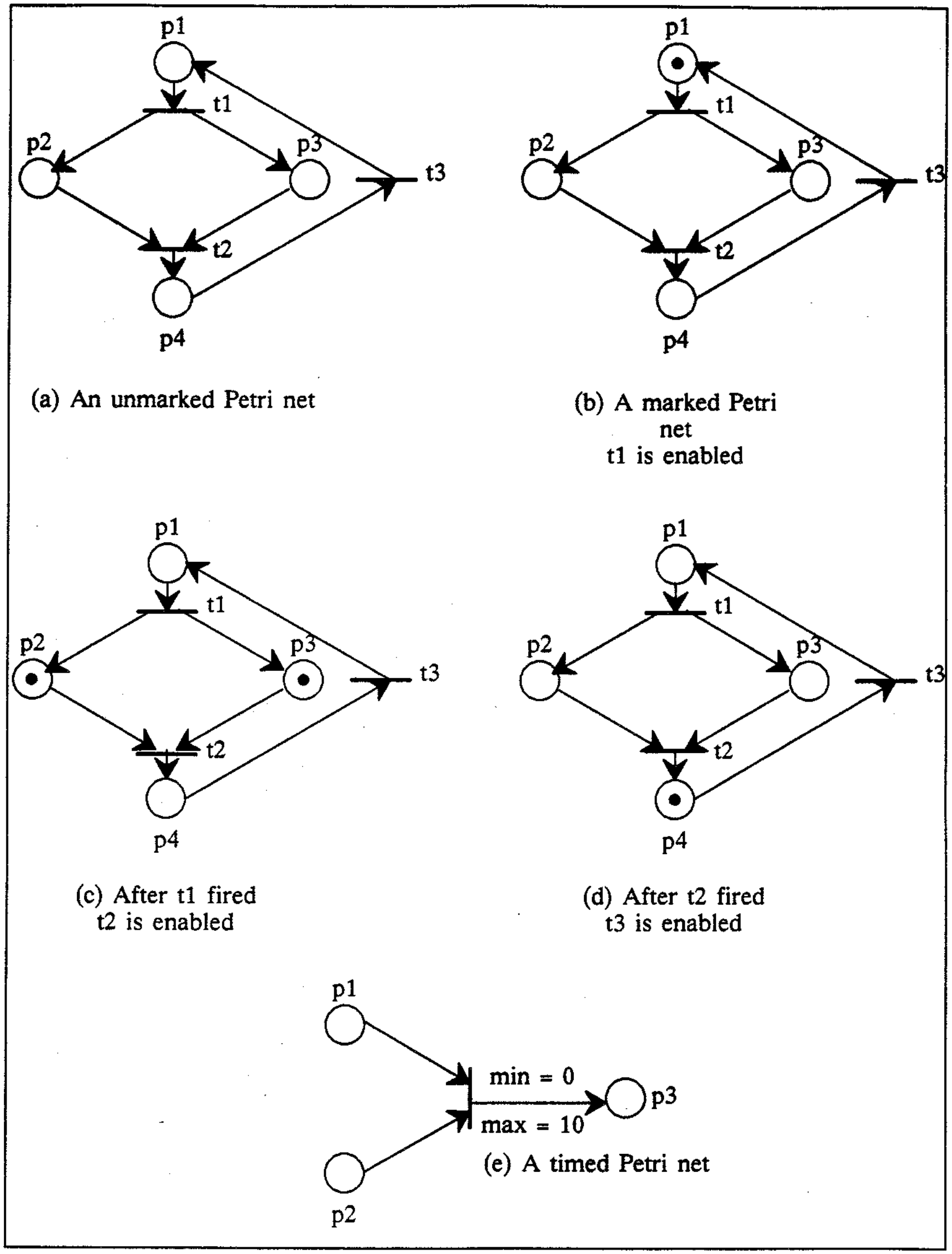
Petri Nets are precise specification tools. They have a well-founded mathematical base and their semantics are well-defined [PN17], [PN21], [PN23], [PN25]. On the other hand, the principle of a Petri Net is so simple that there cannot be any room for misunderstanding.

Petri Nets have very powerful expressive ability. Their computation power is almost equivalent to that of Turing Machine [PN9], such that they have been widely used in modeling computer architectures, computer networks and software systems [PN1], [PN3], [PN4], [PN18], [PN26].

The very simple graphical representation of Petri Nets makes them especially attractive. They are easily understood, learned and used.

A.6.1.3 APPLICABILITY OF PETRI NETS

Petri Nets are especially well suited for *specifying control structures* of computer systems [PN14], [PN20]. The firing rules of marked Petri nets allow all potential asynchronous actions. Thus, a parallel program can be divided into many interacting sub-processes which execute concurrently. Figure A-4 shows the Dijkstra's Five Philosophers Problem.



(a) An unmarked Petri net

(b) A marked Petri net
t1 is enabled

(c) After t1 fired
t2 is enabled

(d) After t2 fired
t3 is enabled

(e) A timed Petri net

Figure A-3: Petri Net Examples

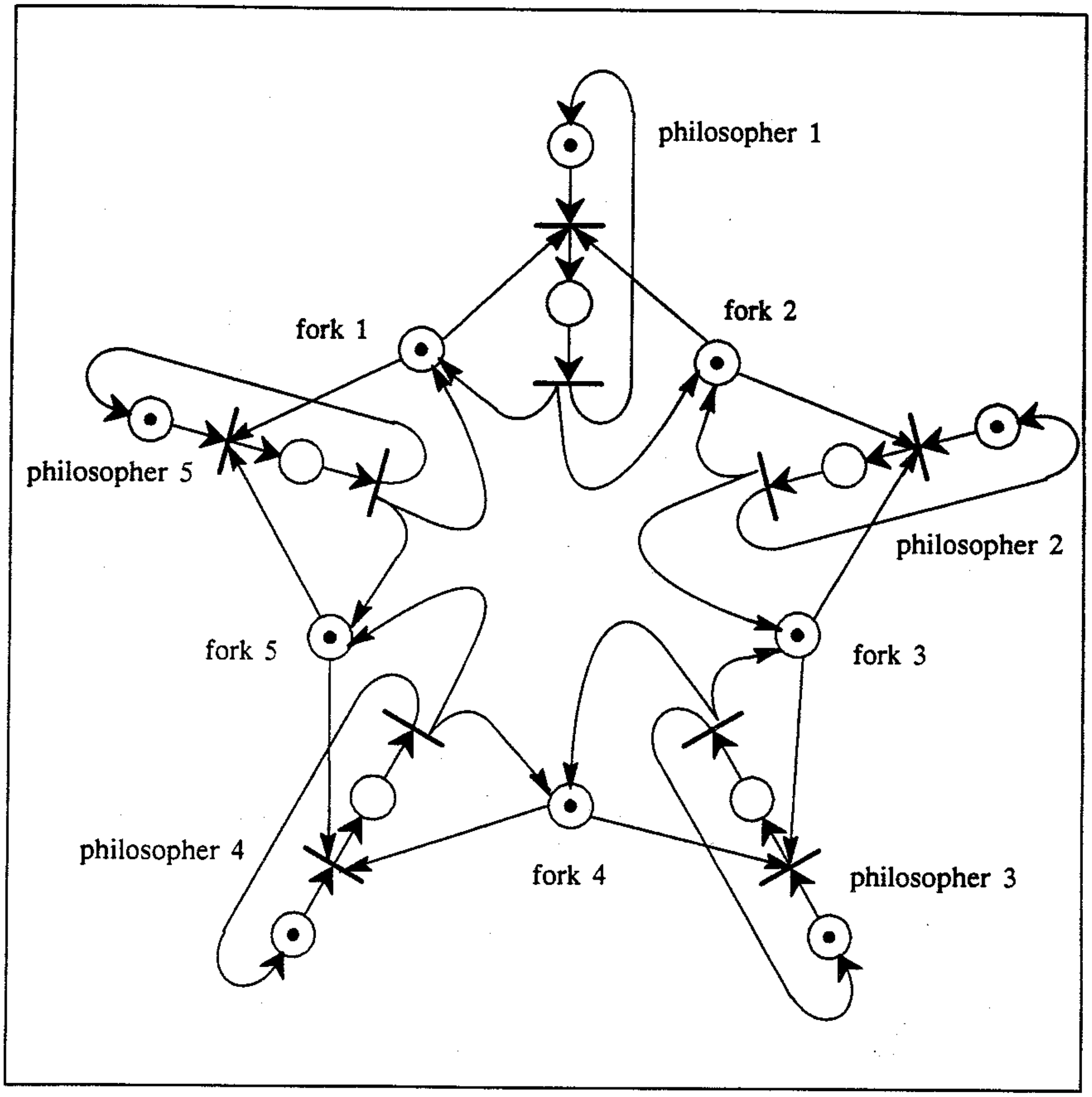


Figure A-4: Dijkstra's Five Philosophers Problem

Petri Nets are very useful in *abstracting the structures* of systems [PN14], [PN24]. Each place and transition can be another complicated Petri Net so that *stepwise refinement* can be made until to user desired level of hierarchy. Example, let

P = P1||P2||...||Pn

be a parallel program, then the top level structure of P can be represented by Figure A-5.

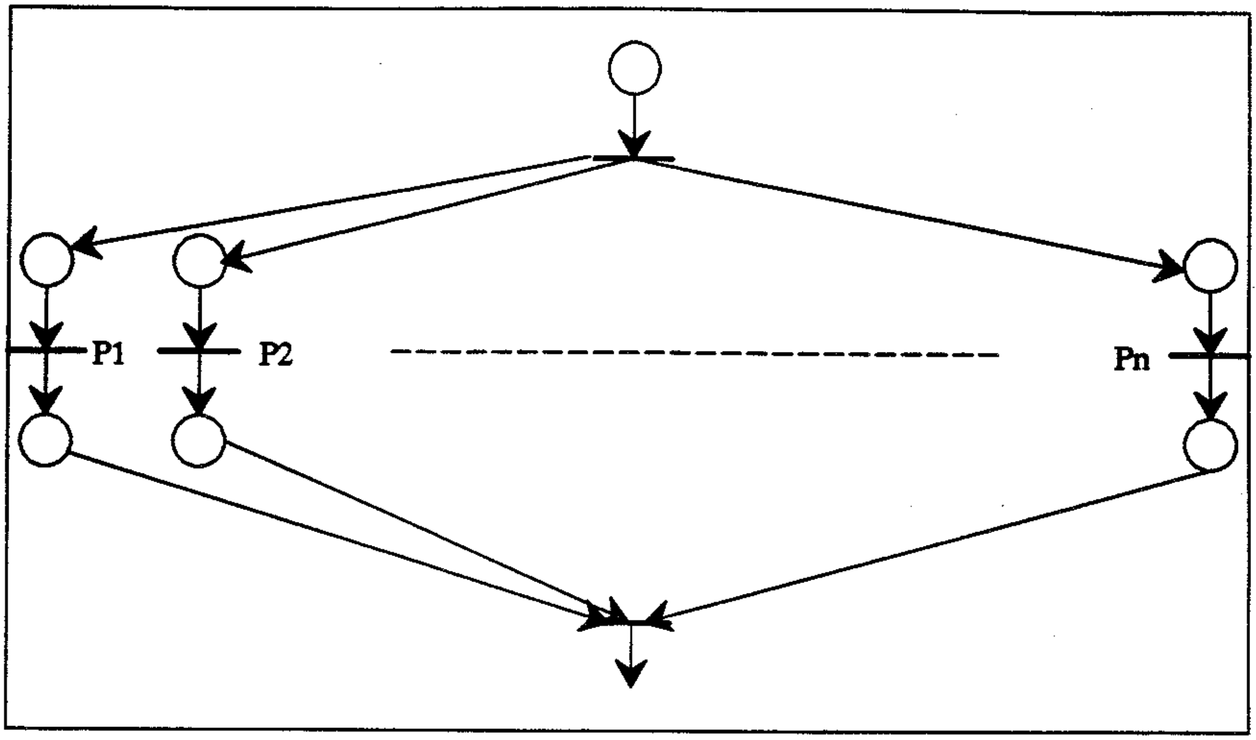


Figure A-5: Parallelism Using Petri Nets

Petri Nets are also excellent tools for analyzing the properties of computer systems. Such properties as liveness (deadlock free) and reachability can be formally verified. There are a variety of theoretical techniques for analyzing Petri Nets [PN12], [PN19].

Timed Petri Nets have another big advantage for *simulating the performance* of *real-time* systems. *Prototyping* models can be built by using Petri Nets which are natural to use and can economize a lot of resources [PN10], [PN13].

A.6.1.4 A HIGH LEVEL SPECIFICATION TOOL

From user to machine, there is a very wide gap. Human languages differ so greatly from machine languages that very few attempts have been made to bridge the gap. Though many specification languages are designed to alleviate the problem, they are generally very low level, too procedural and possess applicabilities which are only appropriate to a small portion of the problem. Hence, a variety of specification languages at different abstract levels is needed to solve this dilemma.

If one believes that a picture saves a thousand words, then a graphical representation is even better than a natural language for human understanding. Petri Nets with a precise graphical representation are *a natural* to be chosen as a top level specification language. One tool is based on this concept, embedded in a Smalltalk 80 environment [PN27].

An Ada program consists of many concurrently executing tasks which communicate each other [PN2], [PN5], [PN15]. Thus, at the top level, only control structure and flow need to be specified. A program consisting of two tasks T1 and T2 is specified by Figure A-6

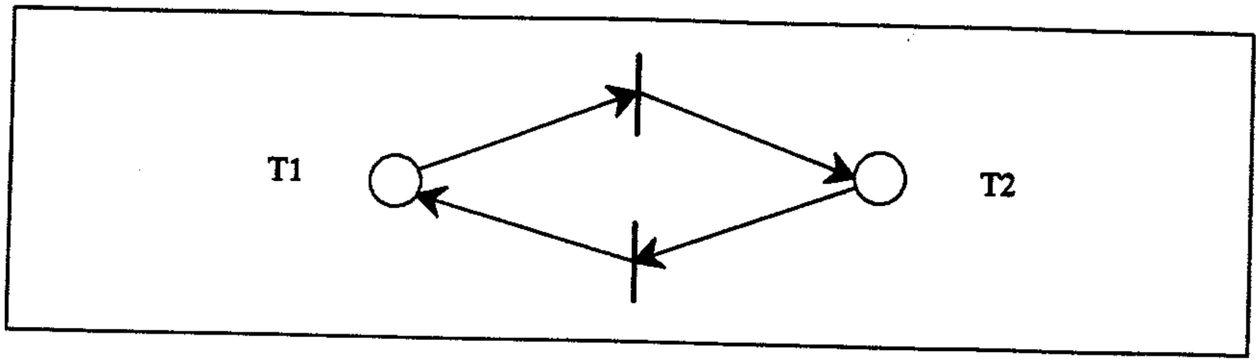


Figure A-6: Control Structure

where each task is represented by a place and the communication is represented by transitions. A further refinement is shown in Figure A-7. The diagram is not only useful

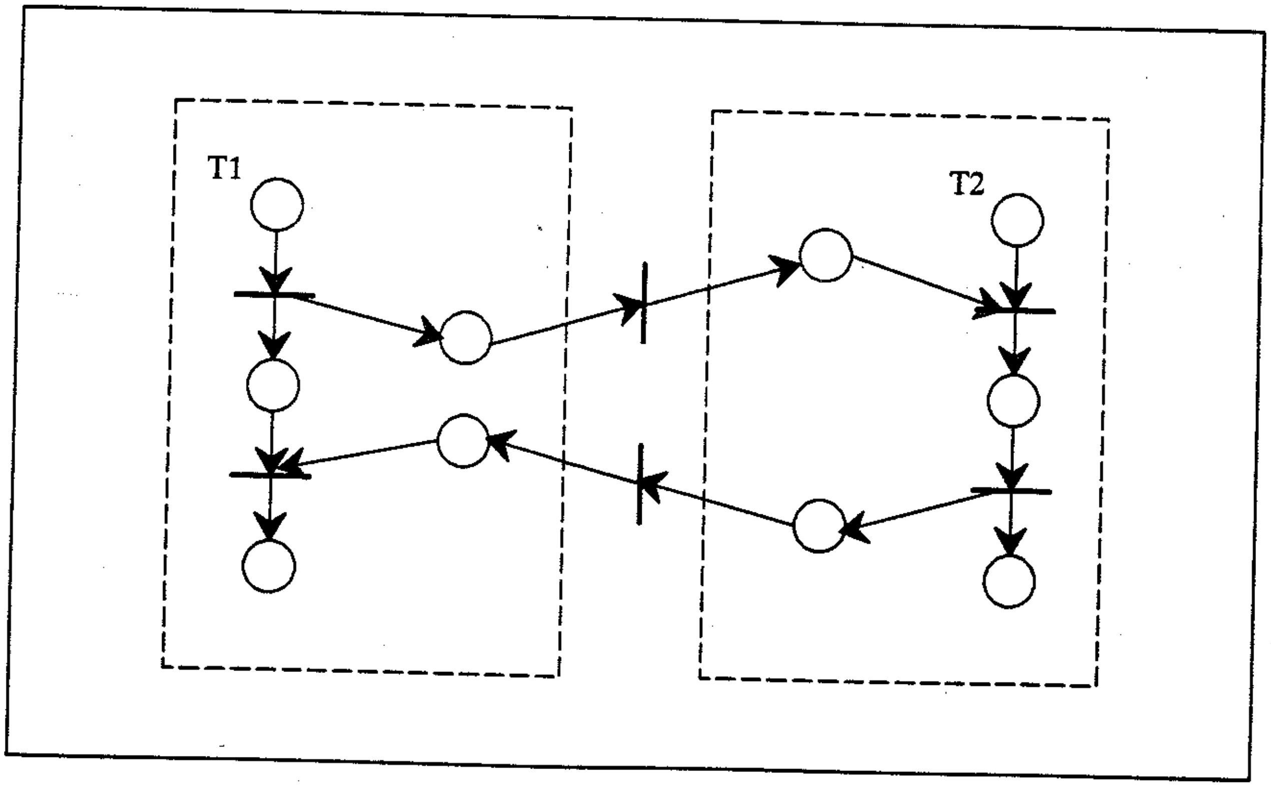


Figure A-7: Rendezvous Mechanism

for programmers to derive implementations, but also understandable to customers; hence the requirements are more likely to be met.

A.6.1.5 BIBLIOGRAPHY -- PETRI NETS

[PN1] Agerwala, T.: "Putting Petri Nets to Work," Computer, Vol. 9, No. 3, Sept. 1979, pp. 85–94.

[PN2] ANSI/MIL–STD–1815A–1983: "Reference Manual For The Ada Programming Language," American National Standards, 1983.

[PN3] Barringer, H.: "Formal Specification Techniques for Parallel and Distributed Systems - A Short Review," Proceedings of the 3rd Joint Ada Europe/AdaTEC Conference (ed. J. Teller), Brussels, 1984, pp. 281–294.

[PN4] Brauer, W. (ed.): "Net Theory and Applications," LNCS, Vol. 84, Springer–Verlag, 1980.

[PN5] Cherry, G. W.: "Parallel Programming in ANSI Standard Ada," Reston Publishing, 1984.

[PN6] Choppy, C. and C. Johnen: "Petrieve: Proving Petri Net Properties With Rewriting Systems," LNCS, Vol. 202, Springer–Verlag, 1985, pp. 271–286.

[PN7] Coolahan Jr., J. E. and N. Roussopoulos: "A Timed Petri Net Method for Specifying Real–Time System Timing Requirements," International Workshop on Timed Petri Nets, Torino, Italy, July 1–3, 1985, pp. 24–31.

[PN8] Datta, A. and S. Ghosh: "Synthesis of a Class of Deadlock–Free Petri Nets," Journal of the ACM, Vol. 31, No. 3, July 1984, pp. 486–506.

[PN9] Hack, M.: "Decidability Questions for Petri Nets," Garland Publishing, 1979.

[PN10] Herzag, O.: "Static Analysis of Concurrent Processes fpr Dynamic Properties Using Petri Nets," LNCS, Vol. 70, Springer–Verlag, 1979, pp. 66–90.

[PN11] Jensen, K., M. Kyng and O. L. Madsen: "A Petri Net Definition of a System Description Language," LNCS, Vol. 70, Springer–Verlag, 1979, pp. 348–368.

[PN12] Leveson, N. G. and J. L. Stolzy: "Safety Analysis Using Petri Nets," IEEE Transactions on Software Engineering, Vol. SE–13, No. 3, March 1987, pp. 386–397.

[PN13] Magott, J.: "New NP–Complete Problems in Performance Evaluation of Concurrent Systems Using Petri Nets," IEEE Transactions on Software Engineering, Vol. SE–13, No. 5, May 1987, pp. 578–581.

[PN14] Maiocchi, M.: "The Use of Petri Nets in Requirements and Functional Specification," *System Description Methodologies* (eds. D. Teichroew and G. David), North–Holland, IFIP, 1985.

[PN15] Mandrioli, D., R. Zicari, C. Ghezzi and F. Tisato: "Modeling the Ada Task System By Petri Nets," Computer Language, Vol. 10, No. 1, 1985, pp. 43–61.

[PN16] Nelson, R. A. et al.: "Casting Petri Nets into Programs," IEEE Transactions on Software Engineering, Vol. SE–9, No. 5, Sept. 1983, pp. 590–602.

[PN17] Nielsen, M., G. Plotkin and G. Winskel: "Petri Nets, Event Structures and Domains," LNCS, Vol. 70, Springer–Verlag, 1979, pp. 266–284.

[PN18] Peterson, J. L.: "Petri Nets," Computing Surveys, Vol. 9, No. 3, Sept. 1977.

[PN19] Peterson, J. L.: "Petri Net Theory and the Modeling of Systems," Prentice–Hall, 1981.

[PN20] Ramamoorthy, C. V. & H. H. So: "Survey of Principles and Techniques of Software Requirements and Specification," Software Engineering Techniques Vol. 2, 1977, pp. 265–318.

[PN21] Reisig, W.: "On the Semantics of Petri Nets," *Formal Models of Programming* (eds. J. C. Neuhold & G. Chroust), IFIP, North–Holland, 1985, pp. 347–372.

[PN22] Rozenberg, G. (ed.): "Advances in Petri Nets 1984," LNCS, Vol. 188, Springer–Verlag, 1984.

[PN23] Rozenberg, G. and P.S. Thiagarajan: "Petri Nets: Basic Notions, Structure, Behavior," LNCS, Vol. 224, Springer–Verlag, 1986, pp. 585–668.

[PN24] Stotts Jr., P. D. and T. W. Pratt: "Hierarchical Modeling of Software Systems with Timed Petri Nets," International Workshop on Timed Petri Nets, 1985, pp. 32–39.

[PN25] Winskel, G.: "Petri Nets, Algebras, Morphisms, and Compositionality," Information and Computation, Vol. 72, No. 3, March 1987, pp. 197–238.

[PN26] Yau, S. S. and M. U. Caglayan: "Distributed Software Design Representation Using Modified Petri Nets," IEEE Transactions on Software Engineering, Vol. SE–9, No. 6, Nov. 1983.

[PN27] Dahler, J., et al: "A Graphical Tool for the Design and Prototyping of Distributed Systems," ACM Software Engineering Notes, Vol. 12, No. 3, July 1987, pp. 25–36.

A.6.2 TEMPORAL LOGIC AS A HIGH LEVEL VERIFICATION TOOL

A.6.2.1 GENERAL CONCEPTS

Temporal logic is a kind of *modal logic* which deals with time concepts [TL35], based on first order logic [TL38] plus some *temporal operators* such as the unary operators:

- *always* (represented by $\Box$),
- *sometimes* (represented by $\Diamond$),
- *next* (represented by $\circ$)

and the binary operator *until* (represented by *U*).

Temporal terms are *first order logic terms* – constants, individual variables and function applications such as f(x1, x2, ..., xn), and $\circ$ t where t is a first order logic term. The meaning of $\circ$ t is the next value of t.

Temporal formulae are constructed from *first order logic formulas* (truth values, propositions, predicates and applications of negation $\neg$, disjunction **or**, conjunction **and**, implication $\rightarrow$, identity $\equiv$, and quantifiers $\forall$ and $\exists$) and the application of the above temporal operators. For example:

$$\Box \; ((x > 5) \textbf{ and } (y < 10))$$

is a temporal formula. A formula without temporal operators is said to be *ordinary* (*static, classic*).

Temporal logic formulas can be interpreted over sequences of program states. where a *state* is a mapping which assigns values to all individual variables. Therefore, temporal logic describes the dynamic properties of a process while ordinary first order logic gives the static interpretation of a formula over a single state.

Let σ : s_0, s_1 , s_2 , ... be an infinite state sequence and σ (k) be a k–shifted sequence: s(k), s(k+1), ... If u, v, w are ordinary first order formulas, then the meanings of temporal operators can be explained as follows:

$\Box$ w holds on σ iff w is satisfied in all states in σ.

$\Diamond$ w holds on σ iff w is satisfied by at least one state in σ.

$\circ$ w holds on σ iff w is satisfied by a $\sigma(1)$

u U v holds on σ iff sometimes v holds and until then u holds continuously.

There are three kinds temporal logic: *linear time temporal logic* [TL3], [TL11], [TL18], [TL21] *branching time temporal logic* [TL6], [TL8] and *interval temporal logic* [TL26], [TL37]. The differences between these systems can be seen in the structures of the state sequence which are allowed. In linear time temporal logic, each state in the sequence can have one and only one successor state. In branching time temporal logic, each state in the sequence can have more than one successor state so that the state sequence can be a tree structure. In interval temporal logic, the temporal formula is interpreted over a specified interval of the state sequence.

A.6.2.2 CHARACTERISTICS OF TEMPORAL LOGIC

Temporal Logic is a very precise formalism. It is based on first order logic, and thus has a well-founded mathematical base [TL35], [TL38].

Temporal Logic has very powerful expressive ability. It can express anything which is definable in first order logic. Its computation power is equivalent to that of first order logic. It has been widely applied in specifying and verifying computer architectures, network protocols, software systems and various program properties [TL4], [TL6], [TL11], [TL12], [TL16], [TL18], [TL20], [TL28], [TL30], [TL31], [TL36].

A.6.2.3 APPLICABILITY OF TEMPORAL LOGIC

Temporal logic is especially well suited for specifying dynamic properties of computer systems such as concurrency and various control structures [TL3], [TL4], [TL11], [TL15], [TL18], [TL20], [TL21], [TL28], [TL30], [TL33], [TL36]. The temporal formulas are so powerful that all potential asynchronous actions can be described.

Temporal logic is directly machine provable. Many program properties such as safety (partial correctness, clean behavior, mutual exclusion, deadlock freedom) and liveness (total correctness, accessibility, responsiveness, fairness) can be specified and verified in the same temporal logic framework [TL22], [TL31].

Interval temporal logic can be used to defining and verifying various constraints of real-time systems [TL16], [TL26], [TL36].

A.6.2.4 A HIGH LEVEL VERIFICATION TOOL

There are two basic ways to specify a software

1. By an abstract program that describes its behavior such as using Petri nets. This kind specification is usually easier to be understood but harder to be verified (proved).

2. By a collection of properties that it must satisfy such as using temporal logic. This kind specification has a big advantage that the specification is directly verifiable (provable) in the same kind of framework.

Therefore, temporal logic is an excellent verification tool. It can be used at top level to specify and verify various properties between processes. If a proof system is constructed, these program properties can be automatically verified (proved).

In the following section, a temporal logic specification of Dijkstra's Five Philosopher Problem is given. *Input* (?) and *output* (!) communication commands of Hoare's CSP [TL14] are used in the definition.

Let P_i be a process for philosopher i ($1 \leq i \leq 5$) and S be a process for forks (synchronizer). Each process P_i communicates with S by four operations:

$S!\mathit{pick}_i$ – pick up fork i

$S!\mathit{pick}_{i\oplus 1}$ - pick up fork $i\oplus 1$

$S!\mathit{put}_i$ – put down fork i

$S!\mathit{put}_{i:\oplus+1}$ – put down fork $i\oplus 1$

($\oplus$ denotes addition modulo 5 and – denotes subtraction modulo 5).

The specification for each process P_i, i=1,...,5 is:

$$\Box\ (((S!\mathit{pick}_i)\ \textbf{and}\ (S!\mathit{pick}_{i\oplus 1})) \rightarrow ((S!\mathit{put}_i)\ \textbf{and}\ (S!\mathit{put}_{i\oplus 1})))$$

The specification for S is:

$$\Box(((P_i?\mathit{pick}_i)\ \textbf{and}\ (P_i?\mathit{pick}_{i\oplus 1})) \rightarrow$$
$$(((\neg\ P_{i-1}?\mathit{pick}_i)\ \textbf{and}\ (\neg\ P_{i\oplus 1}?\mathit{pick}_{i\oplus 1}))\ \mathbf{U}\ ((P_i?\mathit{put}_i)\ \textbf{and}\ (P_i?\mathit{put}_{i\oplus 1})))).$$

A.6.2.5 BIBLIOGRAPHY -- TEMPORAL LOGIC

[TL1] Abadi, M. and Z. Manna: "Nonclausal Temporal Deduction," LNCS, Vol. 193, Springer–Verlag, 1985, pp. 79–88.

[TL2] Barringer, H., R. Kuiper and A. Pnueli: "Now You May Compose Temporal Logic Specifications," Proceedings of the 16th ACM Symposium on Theory of Computing, Washington, 1984.

[TL3] Barringer, H.: "Formal Specification Techniques for Parallel and Distributed Systems–A Short Review," Proceedings of the 3rd Joint Ada Europe/AdaTEC Conference (ed. J. Teller), Brussels, 1984, pp. 281–294.

[TL4] Barringer, H., R. Kuiper and A. Pnueli: "A Compositional Temporal Approach to a CSP-Like Language," *Formal Models in Programming* (eds. E. J. Neuhold and G. Chroust), Elsevier Science Publishers, IFIP, 1985, pp. 207–227.

[TL5] Barringer, H.: "Up and Down The Temporal Way," Computer Journal, Vol. 30, No. 2, April 1987.

[TL6] Clarke, E. M. and E. A. Emerson: "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic," LNCS, Vol. 131, Springer–Verlag, 1981, pp. 52–71.

[TL7] Clarke, E. M., M. C. Browne, E. A. Emerson and A. P. Sistla: "Using Temporal Logic for Automatic Verification of Finite State Systems," *Logics and Models of Concurrent System* (ed. K. R. Apt), 1985, pp. 3–26.

[TL8] Emerson, E. A. and A. P. Sistla: "Deciding Branching Time Logic: A Triple Exponential Decision Procedure for CTL*," LNCS, Vol. 164, Springer–Verlag, 1983.

[TL9] Emerson, E. A. and J. Y. Halpern: "'Sometimes' and 'Not Never' Revisited: On Branching Versus Linear Time Temporal Logic," Journal of the ACM, Vol. 33, No. 1, Jan. 1986.

[TL10] Emerson, E.A.: "Automata, Tableaux and Temporal Logics," LNCS, Vol. 193, Springer–Verlag, 1985, pp. 79–88.

[TL11] Hailpern, L.: "Verifying Concurrent Processes Using Temporal Logic," Springer–Verlag, 1982.

[TL12] Hailpern, L.: "Tools for Verifying Network Protocals," *Logics and Models of Concurrent System* (ed. K. R. Apt), 1985, pp. 57–76.

[TL13] Harel, D.: "First Order Dynamic Logic," LNCS, Vol. 68, Springer–Verlag, 1979.

[TL14] Hoare, C. A. R.: "Communicating Sequential Processes," Communications of the ACM, Vol. 21, No. 8, Aug. 1978, pp. 666–677.

[TL15] ISO/TC 97/WG1–FDT Group/Subgroup C: "Draft Tutorial Document – Temporal Ordering Specification Language," ISO/TC 97/SC 16/WG1N, August 12, 1983.

[TL16] Koymans, R., R. K. Shyamasundar, W.P. de Roever, R. Gerth and S. Arun–Kumar: "Compositional Semantics for Real-Time Distributed Computing," LNCS, Vol. 193, Springer–Verlag, 1985, pp. 167–189.

[TL17] Lamport, L.: "TIMESETS - A New Method for Temporal Reasoning about Programs," LNCS, Vol. 131, Springer-Verlag, 1981, pp. 177-196.

[TL18] Lamport, L.: "Specifying Concurrent Program Modules," ACM TOPLAS, Vol. 5, No. 2, 1983, pp. 190-222.

[TL19] Lamport, L.: "Reasoning About Nonatomic Operations," 10th Annual ACM Symposium on the Principles of Programming Languages," Austin, Texas, 1983, pp. 28-37.

[TL20] Lamport, L.: "An Axiomatic Semantics of Concurrent Programming Languages," *Logics and Models of Concurrent Systems* (ed. K.R. Apt), NATO ASI Series, Vol. F13, Springer-Verlag, 1985, pp. 77-122.

[TL21] Manna, Z. and A. Pnueli: "Verification of Concurrent Programs: Temporal Proof Principles," LNCS, Vol. 131, Springer-Verlag, 1981, pp. 200-252.

[TL22] Manna, Z. and A. Pnueli: "Verification of Concurrent Programs: the Temporal Framework," *The Correctness Problem in Computer Science* (eds. R. S. Boyer & J S. Moore), Academic Press, 1981, pp. 215-274.

[TL23] Manna, Z. and A. Pnueli: "How to Cook a Temporal Proof System for Your Pet Language," 10th Annual ACM Symposium on the Principles of Programming Languages, Austin, Texas, 1983, pp. 141-154.

[TL24] Manna, Z. and P. Wolper: "Synthesis of Communicating Processes from Temporal Logic Specifications," ACM TOPLAS, Vol. 6, No. 1, 1984, pp. 68-93.

[TL25] McLean, J.: "A Complete System of Temporal Logic for Specification Schemata," LNCS, Vol. 164, Springer-Verlag, 1983.

[TL26] Moszkowski, B. and Z. Manna: "Reasoning in Interval Temporal Logic," LNCS, Vol. 164, Springer-Verlag, 1983, pp. 360-370.

[TL27] Moszkowski, B. C.: "Executing Temporal Logic Programs," Cambridge Univ. Press, 1986.

[TL28] Ngnyen, V., A. Demers, D. Gries and S. Owicki: "Behavior: A Temporal Approach to Process Modeling," LNCS, Vol. 193, Springer-Verlag, 1985, pp. 237-254.

[TL29] Pnueli, A.: "The Temporal Semantics of Concurrent Programs," Theoretical Computer Science, Vol. 13, 1981, pp. 45-60.

[TL30] Pnueli, A. and W. P. DeRoever: "Rendezvous with Ada - A Proof Theoretical View," Proceedings of the AdaTEC Conference, Crystal City, 1982, pp. 129-136.

[TL31] Pnueli, A.: "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," LNCS, Vol. 224, Springer–Verlag, 1986.

[TL32] Pnueli, A.: "In Transition from Global to Modular Temporal Reasoning about Programs," *Logics and Models of Concurrent System* (ed. K. R. Apt), 1985, pp. 123–146.

[TL33] Queille, J. P. and J. Sifakis: "Specification and Verification of Concurrent Systems in CESAR," LNCS, Vol. 137, Springer–Verlag, 1982, pp. 337–351.

[TL34] Reynolds, J. C.: "An Introduction to Specification Logic," LNCS, Vol. 164, Springer–Verlag, 1983, pp. 371–382.

[TL35] Rescher, N. and A. Urquhart: "Temporal Logic," Springer–Verlag, 1971.

[TL36] Schwartz, R. L. and P. M. Melliar–Smith: "Temporal Logic Specification of Distributed Systems," Proceedings of the 2nd International Conference on Distributed Computing Systems, Paris, France, 1981, pp. 446–454.

[TL37] Schwartz, R. L., P. M. Melliar–Smith and F. H. Vogt: "An Interval-Based Temporal Logic," LNCS, Vol. 164, Springer–Verlag, 1983, pp. 443–457.

[TL38] Shoenfield, D. R.: "Mathematical Logic," Addison–Wesley, 1967.

[TL39] Wolper, P.: "Specification and Synthesis of Communicating Processes Using an Extended Temporal Logic," 9th Annual ACM Symposium on Principles of Programming Language, 1982, pp. 20–33.

A.6.3 THE AXIOMATIC APPROACH AS A LOW LEVEL VERIFICATION TOOL

A.6.3.1 GENERAL CONCEPTS

The *Axiomatic approach* is based on the principle of Hoare logic [AX13] which utilizes a set of axioms and inference rules. The inference rule usually has the following form:.

$$\frac{H_1, H_2, ..., H_n}{H}$$

where H_1, ..., H_n and H are generalized assertions. The meaning of the formula is that: given H_1, ..., H_n are true, then it can be deduced that H is true.

The meaning of a construct S (expression, statement etc.) is defined by the *pre–condition* P which must be satisfied initially and *post–condition* Q which must be satisfied after the evaluation or execution of the construction. The definition is usually represented by

$$\{P\}\ S\ \{Q\}.$$

For example:

$$\{(x=2) \text{ and } (y=1)\}\quad z := x + y;\quad \{z=3\}$$

meaning that the pre-conditions x=2 and y=1 of assignment statement z:=x+y guarantee the validity of post-condition z=3 after its execution. Thus the meaning of the assignment statement is specified by the assertions.

One of the best known formalisms of this approach is *Predicate Transformer* which was first developed by Dijkstra [AX7] and augmented by Gries [AX12].

A.6.3.2 CHARACTERISTICS OF AXIOMATIC APPROACH

The axiomatic approach has a very well-founded mathematical base. The pre- and post-conditions are basically expressed by first order logic formulas. The axioms and inference rules are or may be confirmed by first order logic. Therefore, the system is very precise [AX1], [AX4], [AX6], [AX11].

The axiomatic approach is very powerful. The axioms and inference rules can be constructed to express anything which can be expressed in first order logic. There are many examples of the practical use of the approach [AX1], [AX2], [AX3], [AX5], [AX9], [AX10], [AX14], [AX17], [AX18], [AX21], [AX27].

The axiomatic approach is basically very simple. Besides the ordinary first order logic symbols and computation and reasoning rules, the pre- and post-conditions are constructed from program variables and propositions. Therefore, they can be used by programmers at very low level [AX8], [AX9], [AX12], [AX13], [AX22], [AX27]. However, the confirmation that the pre- and post-conditions are necessary or sufficient is often beyond their capabilities.

A.6.3.3 APPLICABILITY OF AXIOMATIC APPROACH

The Axiomatic approach is well suited for specifying and verifying sequential program properties. It is particularly useful in proving the partial correctness of programs [AX3], [AX4], [AX10], [AX14], [AX21], [AX25], [AX26].

The Axiomatic approach is abstract. Not only is there no explicit abstract machine or computation sequence concept, but even the concept of state is implicit. The meaning of a software construct is specified entirely in terms of the external observable effect of evaluating or executing it. This can be construed as providing a flexibility in description which can be tuned to the prevailing environment or by lacking a underlying common model, as being difficult to move across boundaries in reusable software environments.

The notation and principle is so simple and easy to be understood that it can be used directly by programmers at very low level.

The Axiomatic description is machine processable. Tools can be built to find some invariants, which are the basis of the verification of the correctness of pre- and post–conditions. Theorem provers can also be constructed to prove the partial correctness of programs defined in this manner [AX19].

A.6.3.4 A LOW LEVEL VERIFICATION TOOL

Like temporal logic, the axiomatic approach specifies the meaning of a program construct by a collection of properties. Thus it is well suited for verification purposes. Further it is easily understood, learned and used. However, this may be one of its fallibilities, since unless combined with a automatic theorem prover its incorrect usage may be masked by its apparent formality! Though easy to write, meaningful pre– and post–conditions are not necessarily easy to generate. Automatic theorem provers can be built when axiomatic approach is used at low level since axiom system and inference rules can generally be found and constructed.

Though the axiomatic approach can be used at high level for specification purposes. It is very hard to find a consistent and complete axiom system. It is very difficult to find concise invariants for large structures. The pre– and post–conditions at high level are either too abstract to be useful or too complicated to be understood due to the lack of compositionality of axiomatic approach [AX20], [AX28]. As a consequence the axiomatic approach is better to be used locally at a low level of detail.

The following example gives the inference rules in axiomatic approach for the if statement: *if* B *then* S1 *else* S2 and the while statement: *while* B *do* S.

The rules therefore define the meanings of the statements.

$$\frac{\{P \text{ and } B\}\ S1\ \{Q\},\ \{P \text{ and not } B\}\ S2\ \{Q\}}{\{P\}\textit{if}\ B\ \textit{then}\ S1\ \textit{else}\ S2\ \{Q\}}$$

where P is the pre–condition and Q is the post–condition.

$$\frac{\{P \text{ and } B\}\ S\ \{P\}}{\{P\}\ \textit{while}\ B\ \textit{do}\ S\ \{P \text{ and not } B\}}$$

where P is the pre-condition.

A.6.3.5 BIBLIOGRAPHY — AXIOMATIC APPROACH

[AX1] Apt, K. R.: "Ten Years of Hoare's Logic: A Survey - Part I," ACM TOPLAS, Vol. 3, No. 4, Oct. 1981, pp. 431–483.

[AX2] Apt, K. R.: "Ten Years of Hoare's Logic: A Survey - Part II: Nondeterminism," Theoretical Computer Science, Vol. 28, 1984, pp. 83–109.

[AX3] Apt, K. R., N. Francez and W. P. de Roever: "A Proof System for Communicating Sequential Processes," ACM TOPLAS, Vol. 2, No. 3, July 1980, pp. 359–380.

[AX4] de Bakker, J. W.: "Mathematical Theory of Program Correctness," Prentice Hall, 1980.

[AX5] Brookes, S. D.: "An Axiomatic Treatment of a Parallel Programming Language," LNCS, Vol. 193, Springer–Verlag, 1985, pp. 41–60.

[AX6] Cook, S. A.: "Soundness and Completeness of An Axiom System for Program Verification," SIAM Journal of Computing, Vol. 7, No. 1, 1978, pp. 70–90.

[AX7] Dijkstra, E. W.: "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," Communications of the ACM, Vol. 18, No. 8, Aug. 1975.

[AX8] Dijkstra, E. W.: "A Discipline of Programming," Prentice–Hall, 1976.

[AX9] Donahue, J. E.: "Complementary Definitions of Programming Language Constructs," LNCS, Vol. 42, Springer–Verlag, 1976.

[AX10] Gerhart, S. L.:" Proof Theory of Partial Correctness Verification Systems," SIAM Journal of Computing, Vol. 5, 1976, pp. 355–377.

[AX11] Goldblatt, R.: "Axiomatising the Logic of Computer Programming," LNCS, Vol. 130, Springer–Verlag, 1982.

[AX12] Gries, D.: "The Science of Programming," Springer–Verlag, 1981.

[AX13] Hoare, C. A. R.: "An Axiomatic Basis for Computer Programming," Communications of the ACM, Vol. 12, No. 10, 1969, pp. 576–583.

[AX14] Hoare, C. A. R.: "Proof of Correctness of Data Representation," Acta Informatica, Vol. 1, No. 1, 1972, pp. 271–281.

[AX15] Hoare, C. A. R.: "Communicating Sequential Processes," Communications of the ACM, Vol. 21, No. 8, Aug. 1978, pp. 666–677.

[AX16] Hoare, C. A. R.: "Some Properties of Predicate Transformers," Journal of the ACM, Vol. 25, 1978, pp. 461–480.

[AX17] Hoare, C. A. R. and N. Wirth: "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica, Vol. 2, 1973, pp. 335–355.

[AX18] Hoare, C. A. R. et al: "Laws of Programming," Communications of the ACM, Vol. 30, No. 8, Aug., 1987, pp. 672–687.

[AX19] Igarashi, S., R. L. London and D. C. Luckham: "Automatic Program Verification I: A Logical Basis and Its Implementation," Acta Informatica, Vol. 4, 1975, pp. 145–182.

[AX20] Lamport, L.: "Hoare Logic of CSP and All That," ACM TOPLAS, Vol. 6, No. 2, April 1984, pp.281–296.

[AX21] London, R. L. et al: "Proof Rules for The Programming Language Euclid," Acta Informatica, Vol. 10, 1978, pp. 1–26.

[AX22] Marcotty, M. and H. F. Ledgard: "A Sampler of Formal Definitions," Computing Surveys, Vol. 8, No. 2, June 1976, pp. 191–276.

[AX23] Milne, R.: "Transforming Predicate Tranformers," *Formal Description of Programming Concepts* (ed. E. J. Neuhold), IFIP, North–Holland, 1978, pp. 31–66.

[AX24] Milner, R.: "A Calculus for Communicating Systems," LNCS, Vol. 92, Springer–Verlag, 1980.

[AX25] Owicki, S. and D. Gries: "Verifying Properties of Parallel Programs – An Axiomatic Approach," Communications of the ACM, Vol. 19, No. 5, Aug. 1976, pp. 279–286.

[AX26] Owicki, S.: "An Axiomatic Proof Technique for Parallel Programs," Acta Informatica, Vol. 6, 1976, pp. 319–340.

[AX27] Page, F. G.: "Formal Specification of Programming Languages: A Panoramic Primer," Prentice–Hall, 1981.

[AX28] de Roever, W. P.: "The Quest for Compositionality: A Survey of Assertion–Based Proof Systems for Concurrent Programs, PartI: Concurrency Based on Shared Variables," *Formal Models of Programming* (eds. E. J. Neuhold and G. Chroust), IFIP, North–Holland, 1985, pp. 157–180.

A.6.4 OPERATIONAL SEMANTICS AS A LOW LEVEL SPECIFICATION TOOL

A.6.4.1 GENERAL CONCEPTS

The *Operational approach* is the earliest formalism that has been developed for specifying the semantics of programming languages and software. It is based on the theory of computation processes (state transitions). The meanings of programs are defined in terms of state transitions of an *abstract machine* which interprets the programs. A machine consists of a set of states (containing two distinguished subsets – the initial states and the end states), a set of transition functions and a function which maps programs and their input data into initial states. The best known formalism of this approach is the Vienna Definition Language (VDL) which has been widely used in various applications [OP5], [OP8], [OP19].

A.6.4.2 CHARACTERISTICS OF OPERATIONAL SEMANTICS

The operational approach is a precise specification formalism, but unlike other formalisms having very well-founded mathematical bases, does not have a mathematically defined semantics. More like a Turing machine or finite automaton, the abstract machine and a set of instructions are constructed to define the meaning of software [OP4], [OP10]. The machine is intended to be so simple that there can be no possible misunderstanding how it works.

The operational approach is very powerful. It can be used to simulate a Turing machine [OP12] and has been widely used in defining the semantics of various programming languages [OP2], [OP7], [OP8], [OP13], [OP16] and software systems [OP20], [OP21]. It has been successfully applied to treat parallelism [OP6], [OP7], [OP16].

A.6.4.3 APPLICABILITY OF THE OPERATIONAL APPROACH

The operational approach is well suited for specifying the semantics of programming languages and software systems. It simulates the actual execution sequences and interprets the meaning of software on an abstract machine. Thus, it is particularly useful for deriving an implementation from the specification [OP11], [OP18], [OP21].

The operational approach is well suited in specifying concurrency, nondeterminancy and real-time requirements.

The approach is very procedural and constructive and resembles the common programming concepts. Thus, it is easy to learn and to use for inexperienced designers and programmers. Further it is machine processable or even directly executable so that it can be used in simulating and prototyping of software systems.

A.6.4.4 A LOW LEVEL SPECIFICATION TOOL

In the operational approach, programs are viewed as computation sequences (state transitions) and the sequence of intermediate states is explicitly given by the definition. Thus, it is very low level and full of details. The meaning of a software is defined by mimicking its behavior on an abstract machine. Hence, the operational approach is not appropriate for verification purposes, except that it can be considered to be a form of N-version programming if the program and its description are developed independently. On the other hand, it is precise and especially useful for deriving a model of an implementation. Consequently the operational approach is well suited as a low level specification tool.

The following is an example of the semantic description of a simple language with two statement forms: the assignment and if statements. In VDL notation, *s–* is used as a component selector and *is–* is used as a predicate identifier.

THE ABSTRACT MACHINE

```
is–ζ =  ( <s–program : is–abstract–program>,
              <s–control : is–control>,
              <s–storage :  is–value–list,
              <s–input    : is–integer–list,
              <s–output   : is–integer–list)
```

The state of an abstract machine in the VDL definition consists of five components:

1. The program : the abstract program;

2. The control : defining the part of the abstract program currently being interpreted;

3. The store : the storage part of the abstract machine;

4. The input file; and

5. The output file;

THE ABSTRACT PROGRAM (SYNTAX)

```
is–abstract–program = is–statement–list.

is–statement        = is–assignment–statement | is–if–statement.
```

```
is-assignment-statement= (<s-variable: is-variable>, <s-expression: is-expression>).

is-if-statement = (<s-compare: is-comparison>,
                   <s-then-clause: is-statement-list,
                   <s-else-clause: is-statement-list).

is-comparison =    (<s-left-operand: is-operand>,
                      <s-operator: is-comparison-operator>,
                      <s-right-operand: is-operand>).

is-expression = is-integer | is-variable | is-infix-expression.

is-infix-expression = (<s-left-operand: is-operand>,
                       <s-operator: is-arithmetic-operator>,
                       <s-right-operand: is-operand>).

is-operand = is-integer | is-variable | is-expression

is-comparison-operator = is-LT
                       | is-EQ
                       | is-NE
                       | is-GT.

is-arithmetic-operator = is-PLUS
                       | is-MINUS.

is-variable = (<s-address: is-integer>)
```

THE INTERPRETER

Some of the evaluation functions are shown below:

```
exec-stmt-seq(sequence) =
    is-<> →    T    → exec-stmt-seq(tail(sequence)) ∧
            exec-stmt(head(sequence)).

exec-stmt(stmt) =
    is-assign-stmt(stmt) → exec-assign-stmt(stmt)
    is-if-stmt(stmt) → exec-if-stmt(stmt).

exec-assign-stmt(stmt) =
    store(s-address.s-variable(stmt), value)
    value: eval-expr(s-expression(stmt))
```

A.6.4.5 BIBLIOGRAPHY -- OPERATIONAL SEMANTICS

[OP1] Anderson, E. B., F. C. Belz & E. K. Blum: "Issues in the Formal Specifications of Programming Languages," *Formal Description of Programming Concepts* (ed. E. J. Neuhold), IFIP, North-Holland, 1978, pp. 1-30.

[OP2] Belz, F. C., E. K. Blum & D. Heimbigner: "A Multi-Processing Implementation Oriented Formal Definition of Ada in SEMANOL," ACM SIGPLAN Notices, Vol. 15, No. 11, 1980, pp. 202-212.

[OP3] Dennis, J.: "An Operational Semantics for a Language with Early Completion Data Structures," LNCS, Vol. 107, Springer-Verlag, 1981, pp. 260-267.

[OP4] Landin, P. J.: "The Mechanical Evaluation of Expressions," Computer Journal, Vol. 6, 1964, pp. 308-320.

[OP5] Lee, JAN: Computer Semantics, Van Nostrand, 1972.

[OP6] Li, W. and P. E. Lauer: "Using the Structural Operational Approach to Express True Concurrency," *Formal Models in Programming* (eds. E. J. Neuhold and G. Chroust), Elsevier Science Publishers, IFIP, 1985, pp. 373-398.

[OP7] Li, W.: "An Operational Semantics for Ada Multitasking and Exception Handling," Proceedings of AdaTEC Conference, Washington, 1982.

[OP8] Lucas, P. and K. Walk: "On the Formal Description of PL/I," Annual Review in Automatic Programming, Vol. 6, 1969, pp. 105-152.

[OP9] Lucas, P.: "Main Approaches to Formal Specifications," *Formal Specification and Software Development* (eds. D. Bjorner & C. B. Jones), Prentice-Hall, 1982.

[OP10] McCarthy, J.: "Towards a Mathematical Science of Computation," IFIP 62 (ed. C. M. Popplewell), 1963, pp. 21-28.

[OP11] Mazaher, S. and D. M. Berry: "Deriving a Compiler from An Operational Semantics Written in VDL," Computer Language, Vol. 10, No. 2, 1985, pp. 147-164.

[OP12] Mejia, L.: "A Proposal for Operational Semantics and Equivalence of Finite Asynchronous Processes," LNCS, Vol. 107, Springer-Verlag, 1981, pp. 387-400.

[OP13] Ollongren, A.: "Definition of Programming Languages By Interpreting Automata," Academic Press, 1974.

[OP14] Pagan, F. G.: "Formal Specification of Programming Languages: A Panoramic Primer," Prentice-Hall, 1981.

[OP15] Plotkin, G. D.: "A Structural Approach to Operational Semantics," TR DAIMI-FN 19, Computer Science Department, Aarhus Univ., 1981.

[OP16] Plotkin, G. D.: "An Operational Semantics for CSP," Proc. of IFIP Working Conf. on Formal Description of Programming Concepts II, North-Holland, 1982, pp. 199-223.

[OP17] Pratt, V. R.: "Using Graphs to Understand PDL," LNCS, Vol. 131, Springer-Verlag, 1981, pp. 387-396.

[OP18] Swartant, W. and R. Balzer: "On the Inevitable Intertwining Specification and Implementation," Communications of the ACM, Vol. 25, No. 7, July, 1982.

[OP19] Wegner, P.: "The Vienna Definition Language," Computer Surveys, Vol. 4, No. 1, 1972, pp. 5-63.

[OP20] Zave, P.: "An Operational Approach to Requirement Specification for Embedded Systems," IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, 1982.

[OP21] Zave, P. and W. Schell: "Salient Features of An Executable Specification Language and Its Environment," IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, 1986, pp. 312-325.

A.6.5 DENOTATIONAL APPROACH AS A SPECIFICATION AND VERIFICATION TOOL

A.6.5.1 GENERAL CONCEPTS

The *Denotational approach (semantics)* was called *mathematical semantics* previously and is based on Lambda Calculus and Scott's *domain theory*. In denotational approach, entities in software are mapped onto mathematical objects (list, tuple, set, function, predicate etc.). In such a definition, three parts are identified:

- syntactic domains,
- semantic domains, and
- semantic functions that map objects in syntactic domains to those in semantic domains.

One important aspect of this approach is that the meaning of a composite structure is composed from the meanings of its components [DN3], [DN5], [DN19].

A.6.5.2 CHARACTERISTICS OF THE DENOTATIONAL APPROACH

Denotational semantics have a well-founded mathematical base in Lambda Calculus and Domain Theory [DN5], [DN31], and thus is a very precise specification method. The methodology is very powerful and can be used to represent anything which is also definable in Lambda-Calculus. It has been very successfully applied in defining the formal semantics of various programming languages [DN1], [DN2], [DN10], [DN19], [DN34], [DN36] and software systems [DN4], [DN11], [DN12], [DN16], [DN21], [DN22].

A.6.5.3 APPLICABILITY

The approach is well suited for specifying and verifying squential software entities [DN6], [DN8], [DN11], [DN13], [DN27] and is especially suited to describe functional relations and recursion. It has also been used in dealing with concurrency [DN4], [DN7], [DN15], [DN16], [DN26]. Since the methodology is precise and mathematical, formal proof and transformation can be undertaken [DN27], [DN36]. Similarly the denotational approach is abstract and compositional so that stepwise refinement and modular design can be utilized [DN11], [DN21]. Moreover, the approach is machine processable and implementations can be derived from the definitions [DN17], [DN22], [DN28], [DN29], [DN35].

A.6.5.4 A SPECIFICATION AND VERIFICATION TOOL

As stated previously, the denotational approach is highly mathematical and was previously named mathematical semantics. The original Oxford notations are very difficult to understand, but syntactic modifications such as the Vienna Development Method (VDM) alleviate the readability problem. While VDM has been proposed as method over the complete development cycle, in our opinion it is still not appropriate to be used at very high level. It is abstract, constructive and very formal so that it is well suited to intermediate level specification and verification methods.

The following example shows a piece (incomplete) of denotational definition of assignment statement and while statement:

Let the semantic domain (state) be the map from ID (identifier set) to VAL (value set):.

$$\delta: \mathrm{ID} \rightarrow \mathrm{VAL}.$$

Denotational semantics define δ:. Let Σ be the set of all states and M[DNs] be the denotation of structure s, then for expression e and statement st, we have:

$$M[e] : \Sigma \rightarrow \mathrm{VAL}$$

$$M[st]: \Sigma \rightarrow \Sigma$$

The denotational semantics for assignment statement and while statement is as following:.

$$M[id{:=}e](\sigma) = \mathrm{assign}(\sigma, id, M[e](\sigma))$$

where assign(σ, id, val) = σ' and

$$\sigma'(x) = \begin{cases} val & \text{when } x = id \\ \sigma(x) & \text{when } x \neq id \end{cases}$$

$$M[\mathit{while}\ e\ \mathit{do}\ s](\sigma) = \begin{cases} M[\mathit{while}\ e\ \mathit{do}\ s](M[s](\sigma)) & \text{if } M[e](\sigma) \\ \sigma & \text{if } \neg M[e](\sigma) \end{cases}$$

A.6.5.5 BIBLIOGRAPHY — DENOTATIONAL SEMANTICS

[DN1] Allison, L.: "Programming Denotational Semantics I," Computing Journal, Vol. 26, No. 2, 1983, pp. 164–174.

[DN2] Allison, L.: "Programming Denotational Semantics II," Computing Journal, Vol. 28, No. 5, 1985, pp. 480–486.

[DN3] Allison, L.: "Practical Introduction to Denotational Semantics," Cambridge University Press, 1986.

[DN4] de Bakker, J. W. and J. I. Zucker: "Processes and the Denotational Semantics of Concurrency," Information and Control, Vol. 54, No. 1–2, July–Aug., 1983, pp. 70–120.

[DN5] Barendregt, H. P.: "The Lambda Calculus – Its Syntax and Semantics," North–Holland, 1981.

[DN6] Bekic, H. and K. Walk: "Formalization of Storage Properties," Lecture Notes in Mathematics, Springer–Verlag, Vol. 188, 1971.

[DN7] Berry, D. M.: "A Denotational Semantics for Shared Memory Parallelism and Nondeterminism," Acta Informatica, Vol. 21, 1985, pp. 599–627.

[DN8] Bjorner, D. and C. B. Jones (eds.): "The Vienna Development Method: The Meta-Language," LNCS, Vol. 61, Springer-Verlag, 1978.

[DN9] Bjorner, D. (ed.): "Abstract Software Specification," LNCS, Vol. 86, Springer-Verlag, 1980.

[DN10] Bjorner, D. and O. N. Oest: "Towards a Formal Definition of Ada," LNCS, Vol. 98, Springer-Verlag, 1980.

[DN11] Bjorner, D. and C. B. Jones: "Formal Specification & Software Development," Prentice-Hall, 1982.

[DN12] Bjorner, D.: "On the Use of Formal Specification Methods in Software Development," 9th Software Engineering Conference, 1987, pp. 17–29.

[DN13] Blikle, A.: "A Metalanguage for Naive Denotational Semantics," Collana Cnet, Vol. 104, 1984.

[DN14] Bohm, C. (ed.): "Lambda-Calculus and Computer Science Theory," LNCS 37, Springer-Verlag, 1975.

[DN15] Broy, M.: "Denotational Semantics of Concurrent Programs with Shared Memory," 1984.

[DN16] Brum, A. D. and W. Bohm: "The Denotational Semantics of Dynamic Networks of Processes," ACM TOPLAS, Vol. 7, No. 4, Oct. 1985, pp. 656–679.

[DN17] Ganzinger, H.: "Transforming Denotational Semantics into Practical Attribute Grammars," LNCS, Vol. 94, Springer-Verlag, 1980, pp. 1–69.

[DN18] Goguen, J. A. and K. P. Ghoni: "Algebraic Denotational Semantics Using Parameterized Abstract Modules," LNCS, Vol. 107, Springer-Verlag, 1981, pp. 292–309.

[DN19] Gordon, M. J. C.: "The Denotational Description of Programming Languages," Springer-Verlag, 1979.

[DN20] Jones, C. B.: "Software Development, A Rigorous Approach," Prentice-Hall, 1980.

[DN21] Jones, C. B.: "Systematic Development Using VDM Approach," Prentice-Hall, 1986.

[DN22] Jones, N. D. and D. A. Schmidt: "Compiler Generation from Denotational Semantics," LNCS, Vol. 94, Springer-Verlag, 1980.

[DN23] Keller, R. M.: "Denotational Models for Parallel Programs with Indeterminate Operators," *Formal Description of Programming Concepts* (ed. E. J. Neuhold), 1978, pp. 337–366.

[DN24] Milne, R. and C. Strachey: "A Theory of Programming Language Semantics," John Wiley & Sons, 1976.

[DN25] Olderag, E. R. and C. A. R. Hoare: "Specification-Oriented Semantics for Communicating Processes," Acta Informatica, Vol. 23, 1986, pp. 9–66.

[DN26] Plotkin, G. D.: "A Power Domain Construction," SIAM Journal of Computing, Vol. 5, No. 3, 1976, pp. 452–487.

[DN27] Polak, W.: "Program Verification Based on Denotational Semantics," Conference Record of the 8th Annual ACM Symposium on the Principles of Programming Languages, 1981.

[DN28] Raskovsky, M. and P. Collier: "From Standard to Implementation Denotational Semantics," LNCS, Vol. 94, Springer-Verlag, 1980, pp. 94–139.

[DN29] Royer, V.: "Transformations of Denotational Semantics in Semantic Directed Compiler Generation," ACM SIGPLAN Notices, Vol. 21, No. 7, July 1986.

[DN30] Schwartz, J.: "Denotational Semantics of Parallelism," LNCS, Vol. 70, Springer-Verlag, 1979, pp. 191–202.

[DN31] Scott, D. S.: "Data Types as Lattices," SIAM Journal on Computer Science, Vol. 5, No. 3, 1976, pp. 522–587.

[DN32] Scott, D. S.: "Domains for Denotational Semantics," LNCS, Vol. 140, Springer-Verlag, 1982, pp. 577–613.

[DN33] Stoy, J. E.: "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory," MIT Press, 1977.

[DN34] Tennent, R. D.: "The Denotational Semantics of Programming Languages," Communcations of the ACM, Vol. 19, No. 8, 1976.

[DN35] Trakhtenbrot, B. A., J. Y. Halpern & A. R. Meyer: "From Denotational to Operational and Axiomatic Semantics for ALGOL-Like Languages: A Overview," LNCS, Vol. 164, Springer-Verlag, 1983, pp. 474–500.

[DN36] Wirsing, M.: "Denotational Semantics of Algebraic Specification Languages," *Formal Models in Programming* (eds. E. J. Neuhold & G. Chroust), IFIP, North-Holland, 1985, pp. 259–284.

A.6.6 ALGEBRAIC APPROACH AS A SPECIFICATION AND TRANSFORMATION TOOL

A.6.6.1 GENERAL CONCEPTS

The *Algebraic approach* is based on Birkhoff's *Heterogenous (many–sorted) algebra* and was first developed by Zilles [AL21], Guttag and ADJ-group [AL15], [AL18]. In the algebraic approach, every software construct is treated as an algebra that consists of a set of values (domains and range), a set of operations upon the domains and equations relating various operations.

A.6.6.2 CHARACTERISTICS OF ALGEBRAIC APPROACH

The algebraic approach has a well–founded mathematical base –– heterogenous algebra and category theory. Its own semantics is well defined and is very precise [AL5], [AL9], [AL15], [AL23]. All total functions can be specified by this methodology which has been widely used in specifying software systems and data abstraction [AL2], [AL3], [AL10], [AL11], [AL12], [AL13], [AL18], [AL24], [AL25].

A.6.6.3 APPLICABILITY

The approach is well suited for specifying and verifying software systems [AL2], [AL3], [AL10], [AL12], is especially appropriate for data abstraction and has been widely used for the description of abstract data types [AL1], [AL4], [AL8], [AL11], [AL13], [AL15], [AL18]. The method is well suited for transformations and correctness proofs. Equational systems and term rewriting systems can be constructed to prove equivalence between formal definitions and to derive low level specifications [AL11], [AL17], [AL20], [AL24], [AL26], [AL28]. Like several other methodologies it is machine processable and direct implementations can be derived from the specification [AL1], [AL18], [AL22]. As with other methodologies, it is abstract so that hierarchical decomposition of specification can be achieved [AL3], [AL5], [AL6], [AL7], [AL27].

A.6.6.4 AN INTERMEDIATE LEVEL SPECIFICATION AND VERIFICATION TOOL

Being highly formal, the method is not appropriate for use at very high level. Similarly, the Algebraic approach is very precise so that transformations and correctness proofs can be carried out formally or even automatically. Therefore, it is well suited as an intermediate level specification and verification method.

The following example gives an outline as to how an algebraic specification appears. The notation is from (Munich) CIP project, though many variants can be found in literature.

The specification defines a SET upon a base domain DATA with the operations add and delete and *hidden functions* isempty and iselem.

```
type SET =
    basedom DATA
    sort set,
    func set empty,
    func (set, data) set add,
    func (set, data) set delete,
    func (set) bool isempty,
    func (set, data) bool iselem,

    isempty(empty)        = true,
    isempty(add(s, x))    = false,

    delete(empty, x)      = empty,
    delete(add(s, x), y)  = delete(s, y)           if x=y,
                            add(delete(s, y), x)   if x ≠ y,

    iselem(empty, x)      = false,
    iselem(add(s, x), y)  = true                   if x=y,
                            iselem(s,y)            if x ≠ y,

    add(add(s, x), y)     = add(add(s,y),x),
    add(s, x)             = x    if iselem(s,x)=true
```

A.6.6.5 BIBLIOGRAPHY -- ALGEBRAIC APPROACH

[AL1] Belkhouche, B. and J. E. Urban: "Direct Implementation of Abstract Data Types from Abstract Specification," IEEE Transactions on Software Engineering, Vol. SE-12, No. 5, May 1986.

[AL2] Broy, M., M. Wirsing and P. Pepper: "On the Algebraic Definition of Programming Languages," ACM TOPLAS, Vol. 9, No. 1, Jan. 1987.

[AL3] Broy, M.: "Algebraic Methods for Program Construction: The Project CIP," *Program Transformation and Programming Environments*, (ed. P. Pepper), Springer-Verlag, 1984, pp.199-222.

[AL4] Burstall, R. and B. Lampson: "A Kernel Language for Abstract Data Types and Modules," LNCS, Vol. 173, Springer-Verlag, 1984, pp. 1-50.

[AL5] Burstall, R. M. and J. A. Goguen: "Putting Theories Together to Make Specifications," Proceedings of 5th International Joint Conference on AI, CMU, Pittsburgh, PA, 1977, pp. 1045–1058.

[AL6] Burstall, R. M. and J. A. Goguen: "Semantics of CLEAR, A Specification Language," *Abstract Software Specifications* (ed. D. Bjorner), LNCS, Vol. 86, Springer–Verlag, 1980, pp. 292–332.

[AL7] CIP Language Group: "The Munich Project CIP: Volume I – The Wide Spectrum Language – CIPL," LNCS, Vol. 183, Springer–Verlag, 1985.

[AL8] Ehrig, H., H.–J. Kreowski, J. W. Thatcher, E. G. Wagner and J. B. Wright: "Parameter Passing in Algebraic Specification Languages," LNCS, Vol. 134, Springer–Verlag, 1982, pp. 322–369.

[AL9] Ehrig, H. and B. Mahr: "Fundamentals of Algebraic Specification – Equations and Initial Semantics," Springer–Verlag, 1985.

[AL10] Ehrig, H. and H. Weber: "Algebraic Specification of Modules," *Formal Models in Programming* (eds. E. J. Neuhold & G. Chroust), North–Holland, IFIP, 1985, pp. 231–258.

[AL11] Flon, L.: "A Unified Approach to the Specification and Verification of Abstract Data Types," IEEE Specification of Reliable Software, 1979, pp. 162–169.

[AL12] Ganzinger, H.: "Modular Compiler Descriptions Based on Abstract Semantic Data Types," LNCS, Vol. 154, Springer–Verlag, 1983, pp. 237–249.

[AL13] Gaudel, M. C.: "Specification of Compilers as Abstract Data Type Representations," LNCS, Vol. 94, Springer–Verlag, 1980, pp. 140–164.

[AL14] Goguen, J. A.: "Abstract Errors for Abstract Data Types," *Formal Description of Programming Concepts* (ed. E. J. Neuhold), IFIP, North–Holland, 1978, pp. 491–526.

[AL15] Goguen, J. A., J. W. Thatcher and E. G. Wagner: "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," *Current Trends in Programming Methodology, Vol. IV* (ed. R. T. Yeh), Prentice–Hall, 1978, pp. 80–149.

[AL16] Goguen, J. A.: "More Thoughts on Specification and Verification," ACM SIGSOFT, Vol. 6, No. 3, 1981.

[AL17] Guttag, J.: "Notes on Type Abstraction," IEEE Transactions on Software Engineering, Vol. SE–6, No. 1, Jan. 1980.

[AL18] Guttag, J. V.: "The Specification and Application to Programming of Abstract Data Types," Ph.D., Thesis, Univ. of Toronto, 1975.

[AL19] Hoare, C. A. R.: "Proof of Correctness of Data Representations," Acta Informatica, Vol. 1, 1972, pp. 271–281.

[AL20] Leszczylowski, J. and M. Wirsing: "A System for Reasoning Within and About Algebraic Specifications," LNCS, Vol. 137, Springer–Verlag, 1982, pp. 257–282.

[AL21] Liskov, B. H. and S. N. Zilles: "Specification Techniques for Data Abstraction," IEEE Transactions on Software Engineering, Vol. SE–1, No. 1, 1975, pp. 7–19.

[AL22] Moitra, A.: "Direct Implementation of Algebraic Specification of Abstract Data Types," IEEE Transactions on Software Engineering, Vol. SE–8, No. 1, Jan. 1982.

[AL23] Mosses, P.: "A Basic Abstract Semantic Algebra," LNCS, Vol. 173, Springer–Verlag, 1984, pp. 87–108.

[AL24] Nakajima, R., M. Honda and H. Nakahara: "Describing and Verifying Programs with Abstract Data Types," *Formal Description of Programming Concepts* (ed. E. J. Neuhold), IFIP, North–Holland, 1978, pp. 527–556.

[AL25] Nivat, M. and J. C. Reynolds(eds.): "Algebraic Methods in Semantics," Cambridge University Press, 1985.

[AL26] O'Donnel, M. J.: "Equational Logic as a Programming Language," MIT Press, 1985.

[AL27] Pepper, P.: "Algebraic Techniques for Program Specification," *Program Transformation and Programming Environments* (ed. P. Pepper), Springer–Verlag, 1984, pp. 231–244.

[AL28] Thatcher, J. W., E. G. Wagner and J. B. Wright: "More on Advice on Structuring Compilers and Proving Them Correct," Theoretical Computer Science, North–Holland, Vol. 15, 1981, pp. 223–249.

APPENDIX B: A SURVEY OF POTENTIAL TOOLS

B.1 PROGRAM DESIGN LANGUAGES

Several program design languages are now available to be used with Ada. Each of these PDL's can help to introduce "formalism" into the software development process. However, the use of any or all of these systems will not result in verifiably correct programs. In the following review of these systems an effort will be made to point out the applicability and limitations of each of these program design languages. Since the effort to introduce formalism into the software development process is in its early stages, plans for future development of these languages will be discussed.

B.1.1 ANNA

ANNA (ANNotated Ada) is a specification language designed for use with Ada that can provide machine processable information about various aspects of the program. [PD6, PD7, PD8, PD9, PD13, PD14] This system is being developed by the Program Analysis and Verification Group at Stanford University. ANNA provided extensions to Ada fall into three categories,

1. generalizations of explanatory constructs already in Ada,
2. additional new kinds of constructs, mostly declarative, dealing with exceptions, context clauses, and subprograms, and
3. additional specification constructs, mainly to specify the semantics of packages and the use of composite and access types.

A significant part of the ANNA language definition is devoted to defining a transformation of specifications into run-time checks. ANNA annotations are added to an Ada program as comment statements, but may generate executable statements if the ANNA compiler is used. The current version of the Transformer, the program which converts ANNA comments into run-time checks, can transform a subset of ANNA. Types and objects that can be annotated have been restricted to scalars only. Object annotations, subtype annotations, statement annotations and subprogram annotations are included in the subset of ANNA which can currently be transformed. Special ANNA attributes have been excluded, and the attribute DEFINED is always assumed to be TRUE. All ANNA expressions are permitted with the exception of those that include quantified expressions or states. Therefore, ANNA membership tests and conditional expressions are included in the subset which can be transformed. Virtual Ada text can be used but the transformer does not include any checks to insure the correctness of this text.

Programs are first put to the transformer. The resulting transformed programs are then compiled by a validated Ada compiler to test for syntactic and semantic correctness. Syntactic errors are found by the ANNA parser, while the transformer detects semantic errors within the annotations. Current diagnostic capability is an error message that reports which annotation was violated, and the line number locating the position of the violation. A robust interface to the transformer is being developed which will initially consist of a major enhancement of the diagnostic information provided during execution of a transformed program, including the location with the source text at which a constraint was violated as well as the location of the corresponding annotation(s). In addition, a symbolic debugger will be added to the system. This will allow the user to query during execution of a transformed program the state of the program that caused an ANNA constraint to be violated. ANNA developers continue to aim at providing a comprehensive system capable of providing support for integrating formal specifications into all phases of software development for Ada.

ANNA is a complex system, which, when complete, will provide a full range of tools to support the development of formally specified and verifiable Ada programs. The initial subset implementations of ANNA appear to have achieved good results. ANNA would be useful now in introducing some formalism into the development of Ada software. It would be possible to implement the ANNA system in a sequence of phases. Each phase would either extend the range of applicability as ANNA is enlarged or it would extend the ANNA concepts of formal specification to another step in the software development cycle. However, since the development of ANNA is likely to extend over the next several years, it does not appear that ANNA alone could be used to verify existing software or to build a library of verified software modules. The use of ANNA is very CPU intensive, not only in the execution time of the resulting software, but also in the execution time of the automated tools. The lack of speed in these tools is due in part to their use of somewhat dated compiler technology, as well as the fact that they are implemented in Ada, where compiler maturity is not at a sufficient level to produce quality code comparable to other languages. This drawback might prevent its application in environments that are unable to provide a sufficient hardware base for development environments, although the benefit of the choice of Ada has allowed transition among available hardware configurations which provide Ada software development environments.

B.1.2 PAMELA

PAMELA [PD1] (Process Abstraction Method for Embedded Large Applications), is a program design method which was developed at Thought Tools, Inc. over the last three years. It features a graphical specification of design and a systematic means for transforming the graphical specification and its elements into Ada program units and their elements.

PAMELA is an Ada-specific software development method. It provides guiding principles and systematic procedures for transforming a software requirements specification into a graphical design and transforming this graphical design into the expressive notations of the Ada programming language. PAMELA is especially suited to real-time, concurrent, and parallel applications, relies on abstract data types for their benefits of information hiding, and introduces abstract processes and abstract data types in a manner natural and understandable to software engineers. PAMELA is object-oriented.

PAMELA is based on a high-level, graphical program design and description language, PAL (for Process Abstraction Language). PAL has icons for modules (abstract data types, abstract processes, abstract procedures, and abstract functions) and for the interfaces between modules. The analyst uses these icons to describe the design by drawing process, subprogram, and abstract data type graphs. These graphs have nodes (the modules) and edges (the calls). AdaGRAPH is a tool that supports PAMELA. It does so in two ways.

- It is a syntax-directed editor for drawing and editing PAMELA graphs and storing them in a data base in an object-oriented representation.

- It is a compiler from PAL into Ada source code to be subsequently compiled by any Ada compiler to target object code.

The PAMELA development cycle contains three steps. Step 1 transforms the software requirements specification into a set of PAMELA graphs. There are three kinds of PAMELA graphs: process graphs (to capture the concurrent and asynchronous processing of the design), subprogram graphs (to capture the sequential processing of the design), and abstract data type graphs (to capture the processing of abstract data types). Step 1 results in a graphical specification (the PAMELA graphs) of a software architecture that satisfies the software requirements. Step 1 identifies the design's major modules (abstract processes, data types, procedures, and functions) and their interconnections.

Step 2 transforms the graphs and their elements to Ada source code. The AdaGRAPH tool performs this step automatically. The first-level process graph is transformed to the

Ada main program. Each lower-level process graph to an Ada package. Each abstract data type is also transformed to an Ada package. This step represents the PAMELA graphs as Ada source code, primarily the declarations, which form the outline of the software.

Step 3 transforms the specifications of the tasks and subprograms to the bodies of these units. At this time the majority of this task must be done manually.

After this review PAMELA appears to be primarily a top and intermediate level design tool. Certainly, the graphics and first level transformation is of value at this level of the design process. In terms of building a more consistent design process this tool would be of great use. However, it does not support the more detailed and maintenance steps in the software development process. Also, it does not address the issue of verification.

B.1.3 BYRON

BYRON [PD4, PD5, PD11, PD12] is a program development language (PDL), designed to provide an efficient mechanism for capturing much of the information produced in the course of engineering a large software system. The BYRON package includes tools that store this information in a structured database and that analyze and extract information from this database to make it available as needed. BYRON was specifically designed to support software development of Ada programs but can be used with other programming languages as well. BYRON does introduce formalism in terms of providing an environment and tools so that consistent and detailed documentation can be built. BYRON does not provide support for program verification and can not provide a basis for introducing mathematical formalism into the software development life cycle. The following information is taken from the BYRON Program Development Language and Document Generator User's Manual. PD4].

The BYRON PDL differs from the Ada language in two important respects. First, it provides a mechanism that associates information about Ada declarations with the declarations themselves. Second, BYRON tools can produce useful output from even the incomplete and inconsistent code that characterizes the early stages of program development. BYRON is used in all phases of the software development process, from specification through maintenance.

BYRON is designed to express information that cannot be expressed in Ada in order to support the software development process. The BYRON PDL includes all of the constructs that are valid in Ada and defines new ones that are used to express this additional information. Comments in Ada begin with "--" and continue to the end of the line. In order to remain compatible with Ada syntax, all BYRON language constructs begin with "--|". Since an Ada compiler treats such a construct as a comment, the same source file can be processed by both the BYRON analyzer and the Ada compiler. The Byron PDL, helps to keep a program and its documentation consistent; since they are kept in the same source file, it is easy to update them together. The special PDL constructs are used only for information that cannot be expressed in Ada. This eliminates redundancy, saving time and removing a potential source of inconsistency.

BYRON provides two general mechanisms for producing documentation based on information stored in the program library. Both of these mechanisms rely on a set of instructions written by document designers to describe the content and format of their documents. In one case these instructions are simple macro interpreted commands. Alternatively these instructions may be Ada source code which is compiled and executed to produce the desired document. The BYRON document generator, BYDOC, processes macro commands to access both BYRON information and Ada semantic information stored in the program library. The macro or "template" language provides a document designer with variables, operators, procedures, data structures, and control statements to express his design. The chief advantages of the template language are that it is easy to learn and use, and that it does not have to be compiled each time a change is made.

There are limitations to using the template language, however, which may make it difficult to describe very complex documents. For example, string operations, arithmetic expressions, and counted loops are some of the features that are no well supported. Rather than adding these features to the template language, which was never intended as a formal programming language, BYRON provides a mechanism for writing Ada programs to access the program library. This mechanism is called the Program Library Access Package (PLAP). Both the PLAP and BYDOC provide useful mechanisms for generating custom documentation. Document designers should weigh requirements for efficiency and complexity before selecting one of these methods.

The tools provided with the BYRON package are:

BYANA Invokes the BYRON PDL analyzer to check a source file for Ada syntax, semantics, and proper use of the BYRON constructs. If there are no errors, it stores a representation of the source file in a program library for subsequent use by other tools

BYDOC Generates a document according to a template file that is specified when the command is invoked. Standard templates are provided, and you can write your own templates to produce exactly the documentation you require. The following list describes the standard templates delivered with the BYRON package:

CALLTREE Produces a report showing the procedures and functions a given program unit calls, and the procedures and functions that call a program unit.

DATADICT Produces a report showing declarations of types, objects, subprograms, packages, tasks, and entries in a selected set of program units, with a brief description of each.

DEPTAB Produces a report showing the dependencies among program units in the library. The report shows what units will be made obsolete if a given unit is recompiled, the order in which program units would be recompiled, and what program units a given program unit "withs" and is "with"ed by, either directly or indirectly.

MAKEC5 Produces a type C5 specification, as required by many military contracts.

USERMAN Produces a report that describes the external interface to a package, subprogram, or task in the program library. This document contains all of the information essential to anyone who wishes to use a predefined program unit.

B.1.4 RAISE

RAISE [PD10] is an acronym for a "Rigorous Approach to Industrial Software Engineering". The aim of the RAISE project is to construct a method and a specification language, based on mathematics, for the development of software in industry, together with a collection of computer based tools supporting the specification language and the method.

A basic concept of RSL (RAISE Specification Language) is the structure. Structures are the building blocks and abstraction units in RSL. Structures constitute the frame in which the RSL entities types, values, variables, operations, and processes are defined.

RSL types can be defined by constructive type expressions similar to the domain equations found in the meta-language of VDM (the Vienna Development Method). Types can be used in the definition of named values. An important kind of value in RSL is the function. A function can be defined in three different ways:

- Explicit definition using the language of value expressions within RSL.
- Implicit definition defining a function by a predicate relating the function result and the parameter.
- Axiomatic definition in which the function is defined through algebraic axioms.

A structure may introduce a state through the declaration of variables, and different instantiations of the state of a structure can be created through copying of such a structure. The variables of a structure are not directly accessible outside the structure. Instead, manipulations of the state of a structure is performed through calls of operations defined in the structure. Operations can be thought of as "functions" with side-effects, and an operation can be defined explicitly using the statements of RSL or implicitly by a predicate.

Parallel activities are described through the introductions of processes in structures. RSL processes are based on CSP [PD3]. Processes communicate through named channels. They can be defined explicitly using RSL parallel combinators or implicitly through a failure assertion which is a predicate describing the allowed sequences of communication.

B.1.5 LARCH

The Larch Project at MIT's Laboratory for Computer Science and DEC's Systems Research Center [PD2] is the continuation of collaborative research into the uses of formal specifications. The project is developing both a family of specification languages and a set of tools to support their use, including language-sensitive editors and semantic checkers based on a powerful theorem prover.

The Larch approach is geared towards specifying program modules to be implemented in particular programming languages. Predicate-oriented interface languages are used to describe the intended behavior of procedures. Abstractions are formulated in the *Shared Language*. Descriptions given in the interface languages are given in terms of those abstractions and might also include descriptions of error-reactions and implementation limits.

Similar in appearance to many algebraic specification languages, the Shared Language can be used for specifying abstract data types, but its focus is on specifying "smaller" entities or properties (such as commutativity, group theory, and generic properties of container-like types). Such entities are expressed as independent, tractable, and reusable building blocks.

The Shared Language offers a simple, syntactic approach to modularization and composition. Units of specifications, called traits, are combined by syntactic inclusion; inclusions can be equipped with renaming rules. Traits are never explicitly parameterized; the renaming mechanism makes any entity of a trait a potential parameter. The meaning of a trait is a first-order theory. It is obtained as the conservative union of the theories associated with included traits and the set of local axioms of a trait. The local axioms are expressed as first-order, quantified equations. The language allows (and the design philosophy encourages) redundant theorems to be stated, thus enabling considerable amounts of consistency checking to be done (possibly by mechanical theorem proving).

B.1.6 DISCUSSION

With the exception of LARCH, each of the program design languages in this section was developed with the programming language Ada in mind. All three of the PDLs described in the preceding summaries integrate some degree of "formalism" into the software development process. Each of them has different points on which they focus. PAMELA is primarily aimed at providing a more consistent, easier to use method of top level design. BYRON is specifically aimed at semi-automatic generation of documentation of Ada source code. Specifically, it can generate documentation reports required for military and other government applications. ANNA is more generally applicable to the entire software development process. Its use supports both more detailed documentation and the inclusion of checking functions useful in the verification process. RAISE is a different approach to the Ada program development problem, typified to some extent by the locale of the effort. RAISE is dependent on the VDM forma description of Ada developed at DDC. LARCH provides an algebraic language for specification. It is fair to say that these PDL's could be considered complementary rather than substitutes for each other. In fact, since both BYRON and ANNA can be used with an underlying DIANA representation of an Ada program it is feasible that they could be used together. Certainly, fully verified Ada programs can not be developed via the use of these tools but a significant level of "formalism" can be introduced by their use.

B.1.7 BIBLIOGRAPHY -- PROGRAM DESIGN LANGUAGES

[PD1] Cherry, G. W.: "PAMELA DESIGNER'S HANDBOOK, Vols. 1&2," The Analytic Sciences Corp., 1986.

[PD2] Guttag, J. V., J. J. Horning and J. M. Wing: "Larch in Five Easy Pieces," Digital Systems Research Center, July 1985, 113 pages.

[PD3] Hoare, C. A. R.: "Communicating Sequential Processes," Communications of the ACM, Vol. 21, No. 8, Aug. 1978.

[PD4] Intermetrics: "BYRON Users Manual," IR-MA-781, Dec. 1986, 74 pages.

[PD5] Illinois Institute of Technology Research Institute: "Ada Verification System (AVS) Studies," Draft Final Report, July 1987, 98 pages.

[PD6] Krieg-Bruckner, B. and D. C. Luckham: "ANNA: Towards a Language for Annotating Ada Programs," Proceedings of the ACM SIGPLAN Symposium on the Ada Programming Language, 1980.

[PD7] Luckham, D. C., F. W. von Henke, B. Krieg-Bruechner and B. Owe: "ANNA A Language for Annotating Ada Programs: Preliminary Reference Manual," 1984, Stanford University.

[PD8] Luckham, D. C. and F. W. von Henke: "An Overview of ANNA, a Specification Language for Ada," IEEE Software, Mar. 1985, pp. 9-22.

[PD9] Luckham, D. C., R. Neff and D. Rosenblum: "An Environment for Ada Software Development Based on Formal Specification," Computer Systems Laboratory Technical Report, CSL T. R. 86-305, Stanford University, 1986.

[PD10] Meiling, E.: "A Spreadsheet Specification in RSL - An Illustration of the RAISE Specification Language," Dansk Datamatik Center.

[PD11] Meyer, B.: "On Formalism in Specifications," IEEE Software, Jan. 1985.

[PD12] Nyberg, K. A., A. A. Hook and J. F. Kramer: "The Status of Verification Technology for the Ada Language," IDA Paper P-1859, Institute for Defense Analyses, July 1985.

[PD13] Rosenblum, D.S.: "A Methodology for the Design of Ada Transformation Tools in a DIANA Environment," IEEE Software, Mar. 1985, pp. 24-33.

[PD14] Sankar, S., D. Rosenblum and R. Neff: "An Implementation of ANNA," Proceedings of the Ada International Conference, Paris, May 1985.

B.2 FORMAL SOFTWARE ASSISTANTS

Because of the ever increasing size and operational criticality of computer systems, design verification and formal proofs of correctness are being incorporated in the software development life cycle. More specifically, design verification is the process of showing that the design of a system expressed in a specification language satisfies the requirements of the system [TP22]. Tools which will support the design verification process are currently being developed. In the following the status of a number of these systems is summarized.

B.2.1 AFFIRM

A major concern of many recent research papers has been the development of specification and verification methods that can take advantage of abstraction and hierarchical structure.

AFFIRM [TP13, TP19, TP21] is being developed as an interactive system for specification and verification. The AFFIRM system accepts data abstractions and programs in a specification language based on the algebraic axioms method and in a programming language based mainly on Pascal. The specification language provides a facility for initially defining and successively refining a characterization of a user's problem. Specifications of data abstractions can be tested for consistency and completeness, and specifications which pass these tests become part of the system data base. The AFFIRM verification condition generator supports the standard inductive assertions method and the subgoal assertions method.

AFFIRM also contains a natural deduction theorem prover for interactive proof of verification conditions and properties of data abstractions. AFFIRM is capable of organizing large specifications and collections of axiomatic and derived properties in an on-line data base for easy retrieval. The AFFIRM system has been successfully used to specify and prove the verification of a symbol table abstraction and its implementation by a stack of hash tables, the correctness of an implementation of Queues in terms of circular lists, and rules for types such as sequences, sets, and trees. Continued development in the areas of axiomatic methods for top-level modeling of systems, requirements analysis, and reusability are needed.

AFFIRM has been shown to be effective at interactive proof and verification for relatively small examples. However, its development is not to the point where it would be appropriate as a tool to help formalize the specifications for or prove the correctness of

Ada programs. Certainly, it might be used to develop a library of "verified" software modules", but even in this use a translation from the pascal-like AFFIRM programming language to Ada would have to be done.

B.2.2 PROLOG

A current research effort involves the reimplementation of the AFFIRM Specification and Verification System [TP11] using Prolog. Prolog "programs/data bases" are sequences of rules and facts, with a backtracking interpreter that unifies goals with the program and data bases producing as many solutions as are feasible and requested. The attractiveness of Prolog is its unification of formalisms such as relational data and production systems. However, program development environments for Prolog are primitive at best and modularity and other structural issues are only starting to be addressed. The new implementation of AFFIRM has been successfully used to prove a previously used benchmark set of lemmas. Performance is comparable to the old InterLisp implementation but has reduced resource requirements. The new Prolog may provide impetus to solve problems in the verification field because of its direction towards use of new architectures, unifying formalism of "logic programming", and the ability of Prolog to naturally express models of problems and organize knowledge.

B.2.3 GYPSY

The following description of the Gypsy Verification Environment (GVE) is taken from "Using the Gypsy Methodology" [TP14].

The Gypsy [TP1, TP14, TP22, TP25] methodology is an integrated system of methods, languages, and tools for designing and building formally verified software systems. The methods provide for the specification and coding of systems that can be rigorously verified by logical deduction to always run according to specification. These specification, programming, and verification methods dictated the design of the program description language Gypsy. Gypsy consists of two intersecting components: a formal specification language and a verifiable, high level programming language. These component languages can be used separately or collectively. The most important characteristic of Gypsy, however, is that it is fully verifiable. The entire Gypsy language is designed so that there exist rigorous, deductive proof methods for proving the consistency of specifications and programs. The methodology makes use of the Gypsy Verification Environment (GVE) to provide automated support. The GVE is a large interactive system that maintains a Gypsy program description library and provides a highly integrated set of tools for implementing the specification, programming, and verification methods.

The Gypsy methodology may also be applied strictly to the design phase of system development. For example in certain applications, particularly in the security domain, it is considered desirable to prove that a system's specifications possess specific properties. In Gypsy these properties would typically be stated as lemmas or as the specifications on an abstract data type, and the verification of the design would consist of demonstrating that the high level specifications satisfy these lemmas and type specifications.

The effective range of application of the methodology depends on the applicability of Gypsy to a particular problem. Gypsy is suitable for a wide range of general and systems programming applications. Gypsy was derived from Pascal and retains much of the wide applicability of Pascal. One major exception is the absence of floating point in Gypsy. Gypsy, however, does have major facilities for exception handling, data abstraction, and concurrency that are not present in Pascal. During its development, the methodology has been used successfully in several substantial experimental applications. These include message switching systems, selected components of an air traffic control system, communication protocols, security kernels, and monitoring of inter–process communication.

The most recent enhancements to the system have been extensions to the theorem prover. Pending enhancements include, a new data flow analysis tool, an updated algebraic simplifier, and extensions to the theorem prover which will allow sequential termination and the well–formedness of non–mutually recursive specification functions to be proven.

B.2.4 SDVS (STATE DELTA VERIFICATION SYSTEM)

This proof mechanism is designed to be an interactive program which assists in the definition and verification of microcode. SDVS provides a formal language which is a classical (unquantified) first–order predicate calculus which can be appended to the description of a microcode. Each definition provides a set of pre– and post–conditions for a piece of microcode which can then be verified through the use of a proof system. In 1985 [TP17] the SDVS was not complete, though several proofs were exhibited. It would be hoped that by 1987 these have been added.

B.2.5 INA JO

Methods for decomposing large conjectures into smaller ones in order to make their proof easier and for limiting the amount of reproving that occurs when a specification is modified are critical parts of the verification process. The FDM (Formal Development Method) is a method of arriving at a formal specification of a system which is verifiably guaranteed to meet requirements that are also stated in the specification. Two of the major obstacles in the iterative process of specification and verification are the size of the conjectures to be proved and the major reproving required whenever a specification is modified. The Ina Jo processor [TP2] presented can decompose specifications written in the Ina Jo specification language into a number of conjectures to be proven as theorems. Proofs are accomplished by a human prover utilizing the ITP (interactive theorem prover). A new proof manager is proposed which will be able to determine what reproving is required when a change is made. The proof manager and and the simplifier are not yet complete. The ITP and the processor still require significant modifications. The Ina Jo system is being developed at the University of California by Daniel M. Berry.

B.2.6 VERUS

VERUS [TP23] is a design verification system developed by Gould Software Division. VERUS is a general purpose system, but its primary intended application is to specify and prove security properties of systems modeled as state machines. The VERUS language is a dialect of the predicate calculus adapted to a software engineering environment. It includes parser directives, types, definitions, and proof directives. Theorems are stated and proved as an integral part of the specification. Each theorem is stated in a proof outline, which includes supporting statements and proof directives to guide the theorem prover. The user composes proof outlines with a standard text editor and interacts with the prover directly via the parser.

B.2.7 ABEL

ABEL (Abstraction Building, Educational Language) [TP11, TP14] is designed to be an integrated system for specification and programming. It contains a language for constructive (algebraic) specification centered on the concepts of types and functions, which can be viewed as an applicative programming language. For the purposes of specification and reasoning, a version of typed first order predicate calculus that utilizes partial functions is included, called Weak Logic, along with mechanisms for non-constructive axiomatics. These language elements are "non-executable," but are supported by software for syntactic processing as well as certain kinds of semi-mechanized reasoning. All language constructs can be combined in writing imperative texts in terms of non-executable concepts, either for specification purposes or as a step in a semi-formalized program development process. They may also be combined in decorating imperative executable programs with pre- and post-conditions and other assertions for Hoare-style reasoning. ABEL is still being developed as a teaching tool at the Institute of Informatics at the University of Oslo primarily by Ole-Johan Dahl and Olaf Owe.

B.2.8 EVES

The principal goal of the EVES project [TP10, TP14] is the development of a program verification system which will usefully support the production of formally verifiable code written in a dialect of Euclid. Euclid is a Pascal-like language which has been modified in ways which have decreased the difficulty of deriving a Floyd-Hoare axiomatization of the language. Research and development are continuing on the Euclid language, the specification system and the theorem prover. EVES is being developed at I.P. Sharp Associates by Dan Craigen, Mark Saaltink and their associates.

B.2.9 HDM

The Hierarchical Development Methodology (HDM) [TP18, TP22] and its associated tools and specification language (known as SPECIAL), which were developed during the 70's have been widely used for the specification and verification of secure systems. A new specification and verification environment that is substantially different from the old HDM is being developed. New features included in the system are:

- Specifications in first order predicate calculus with second order capability,
- Strong type checking with overloading,
- Parameterized modules with semantic constraints,
- User interface based on multi-window screen-editor,
- Theorem proving by reduction to propositional calculus, with decision procedures for common theories,
- Hoare sentences and code proof, and
- Multilevel security (MLS) checking by information flow analysis.

In contrast to systems which seek to automate the theorem proving process, this verification system is designed to provide support for the expert user. In order to prove a theorem, the user cites instances of axioms and previously-proved theorems from which the result follows by propositional reasoning. The user is expected to provide appropriate substitutions for the existentially quantified variables of the conclusion and for the universally quantified variables of the formulae cited as premises. The prover reduces the user's proof from first order into propositional calculus. The reduced formula is submitted to the underlying prover. The new system includes an information flow analyzer similar to the one used with SPECIAL.

B.2.10 VDM

The Vienna Development Method [TP3] was developed by D. Bjorner, W. Henhapl, C.B. Jones, and P. Lucas. It is a systematic approach to the development of large computer software systems. The three steps in the method are Requirements, Formal Definition, and Development. Between each step a justification must be provided. The aim of a justification is to document why the proposed solution is believed to fulfill its specification. Such justifications can be presented at different levels of formality. Decomposition, correctness arguments, and the use of abstraction are key points in the methodology.

A meta-language is used to formally describe specifications and to facilitate verification. All of the constructs of the meta-language are precisely defined. The meta-language supports a denotational representation.

VDM started as a means to systematically develop a compiler from a language definition. However, it has been shown to have a broad range of application.

B.2.11 PVS

The Practical Verification System (PVS) [TP26] is based on transformational verification. It involves transforming specifications at one level of abstraction into specifications at a lower level of abstraction and/or transforming specifications into programs. The philosophy assumes that a programmer who correctly understands his assignment and reliably employs sound implementation strategies will produce a correct program. The system must contain meta-theorems which guarantee that if the system successfully applies a transformation, then that transformation is *mean refining*. In the PVS system, the user supplies design requirements in the form of an initial specification, written in the user's domain of discourse. This user-supplied information relies on high level concepts stored in the PVS database, which contains formalized information about programming and related subjects. The user then supplies transformations in a number of successive steps. In parallel with this, the verification system executes these transformations, obtaining a series of successively lower level specifications, and finally, a program. Since the transformation between each successive specification is proved to be meaning refining, the result of these transformations is actually a verified program. Development of the major modules in the PVS system is under way, the major problem is seen to be the explicit reliance on automatic theorem proving. Improvements in this area over the "state-of-the-art" are needed to build an effective efficient system.

Like SDVS, PVS was in the process of implementation in 1985 when presented at VERkshop III, with the goal of providing a transformational assistant for the development of programs. The user provides a set of design requirements which are to be successively transformed into a functional description, a detailed design and a source code program. Using a data base which contains "formalized information about programming and related subjects", the user must execute the transformational refinements in accordance with predefined strategies thus avoiding mistakes of application. As the user inputs the uppermost level specification, it is checked for syntactic correctness by a parse translator, and for semantic validity by a command verifier, the main component of which is a theorem prover. Once each transformational step has been verified, the command interpreter performs the specified steps and the print translator visually provides user with the results. [TP26] states that "work is currently under way to use the existing parse and print translators as a basis for an interactive user environment ... for more-or-less arbitrary interface languages [and] careful reformulation of the formal basis of the internal logic is being followed by a reformulation of Pascal. Additions to the database will include definitions of ... Ada, C, Euclid, Gypsy and/or LISP."

B.2.12 BOYER-MOORE THEOREM PROVER

The Boyer-Moore Theorem Prover (THM) is unique among tools used in the formal development process in that it has more often been applied in the realm of fundamental mathematics, rather than program verification. The theorem prover was originally developed to support the proof checking needs of the Hierarchical Development Method (HDM) from Stanford Research Institute. However, it branched out from under HDM and began to be used in a number of other applications. In [TP5], examples of its use include application to the proof of verification conditions for FORTRAN [TP6], prime factorization [TP7], Fermat's Little Theorem, the unsolvability of the halting problem [TP8], and Godel's incompleteness theorem [TP24].

THM uses the Boyer-Moore logic for expression of theorems, rather than a particular programming language semantics. This logic is a quantifier-free first order logic, similar to LISP [TP4]. Theorems are presented in the language of this logic, and then submitted to the theorem prover, which acts as a high level proof checker. The individual

4. While complete verification may be impossible with existing techniques, partial verification or validation, involving the proof of one or more properties of a module or program, is a useful indicator in providing a guide of correct behavior.

Tools: To perform validation of a module of code it is suggested that a process similar to the compilation of a module of code be used. The syntactic specification of a piece of code should be validated before that piece of code's body. This allows other modules that call this module to be validated before the module is fully validated. The eventual validation of the body includes maintaining the constraints defined by the syntactic specifications.

Facilitating modular validation, an important construct with MAVEN, is the program library. Syntactic specifications are stored in the program library where they are accessible to module bodies being validated. Semantic specifications are stored in a validation library. The semantic specification for a module consists of a set of pre/postcondition pairs, one for normal termination and one for each exception that may be raised.

To facilitate consistency within the verification process, MAVEN imposes restrictions on the order in which units may be verified. Specifically a module's semantic specification must be entered into the validation library before any other module that references the module, or the module itself, may be considered verified. The order of verification of individual bodies or individual specifications is not considered, only that a specification must be verified before a separate body that refers to it may be verified. Similar restrictions apply to the re-validation of a program after modifications are made.

While the most powerful tools within MAVEN are the libraries (validation and program), there are more verification tools within MAVEN. Some of these tools are specification writer's assistant, a software retrieval tool and a suite a testing tools. The specification writer's assistant is a knowledge-based tool which, through interaction with the user, produces formal specifications. The software-retrieval tool implies validation upon reusable units of code assuming two conditions hold true:

1. The preconditions given in the design imply the corresponding preconditions of the reusable unit.

2. The postconditions given in the design imply the corresponding postconditions of the reusable unit.

submitting the theorem has the opportunity to "guide" the theorem prover to follow particular heuristics in the attempt to prove the correctness of the theorem.

The major research effort in developing THM has been the delineation of various heuristics to mechanize mathematical induction. Other heuristics include selection of axioms, definitions and lemmas for inclusion in proofs, and generalization of a problem to a greater level [TP9].

B.2.13 MAVEN

SofTech has proposed a set of requirements for tools to support the formal verification and validation of Ada programs on a modular basis. This set of requirements is known as the Modular Ada Validation Environment (MAVEN). This project is not being implemented and is meant to serve as a set of guidelines for an Ada Program Support Environment (APSE) that is to support formal verification and validation. SofTech makes the distinction between verification and validation by defining them as such: verification pertains to the formal proof of a piece of code through mathematical techniques, validation pertains to the determination of a level of confidence in the software in question through formal or informal methods.

Requirements: The functionality of MAVEN is based on the establishing of several requirements for the validation of Ada programs:

1. Formal proofs should be implementation independent. The behavior of a specific environment that contains a validated Ada compiler should not affect the performance of MAVEN.

2. The functional performance of MAVEN should be modular in nature. The modification of a modular section of a program should not require the re-validation of other modules of the same program as long as certain pre/postconditions of the modified module are not changed.

3. Separate modules of a program may be validated by different techniques. Different modules lend themselves to different validation techniques. While one module may be formally verified without complications, another module may require a less rigorous technique be used. This difference in techniques should not affect the overall validation of the software. Some of the possible techniques include formal and informal proof techniques, code walkthroughs, unit testing, and historical acceptance based on trusted performance.

Effects on the software life cycle: While MAVEN is mostly applicable to the validation of software, it provides support for a wide variety of topics over the entire life cycle. Only those topics which are directly related to verification will be discussed here. Under MAVEN, verification is not limited to a single do–or–die process, support is given to verification of software throughout the life–cycle. The specification writer's assistant is useful in defining formally stated verifiable requirements. In addition, MAVEN plays four roles in the design phase:

1. The validation library is used for storage of semantic specifications of each design module. Again the specification writer's assistant is used here.

2. During top level design module design plans are formulated and stored in the validation library.

3. MAVEN contains catalog of reusable software including modules previously placed there by the user. These modules are most easily integrated if their formal specifications are stored in a library which is linked to the actual code through automated retrieval. As previously stated, re–validation of this code is not required if the syntactic specifications validate within the scope of the new program.

4. In the same way that standard Ada code is verified an Ada Program Design Language (PDL) is subject to validation through MAVEN. This enables the programmer to prove that top–level algorithms correctly meet the system specifications.

Logically, all verification techniques applicable to top level design are also applicable to intermediate and low level designs.

MAVEN may be used to assure that, when modifications are made to the program for maintenance, the initial specifications are not validated

B.2.14 DISCUSSION

A comglomerate of tools would provide an overlap of capabilities which could be tailored to serve the needs of the formal life cycle. The Software Productivity Consortium has already chosen to use the BYRON system as a basis of program development, and thus it might be appropriate to investigate the means by which additional functionalities can be added to that system to provide a broader toolset supportive of the formal life cycle. Some diagrammatic approaches to the top level of design have already been incorporated into BYRON which could well be adapted and modified to operate over Petri Nets. Similarly at a lower level, the comments in an Ada program can be used to incorporate assertions regarding the properties of the program being described. This development is intended to be a part of the further woek proposed in the main part of this report.

B.2.15 BIBLIOGRAPHY -- FORMAL SOFTWARE ASSISTANTS

[TP1] Ambler, A. L. et al.: "Gypsy: A Language for Specification and Implementation of Verifiable Programs," Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol. 12, No. 3, 1977.

[TP2] Berry, D.M.: "An Ina Jo Proof Manager," ACM Software Engineering News, Vol. 10, No. 4, Aug. 1985.

[TP3] Bjorner, D., W. Henhapl, C. B. Jones, and P. Lucas: *The Vienna Development Method: The Meta-Language*, Springer-Verlag, New York, 1978.

[TP4] R. S. Boyer and J S. Moore: "A Computational Logic", Academic Press, New York 1979.

[TP5] R. S. Boyer and J S. Moore: "Proof-checking, Theorem-proving, and Program Verification," Technical Report 35, Institute for Computing Science, The University of Texas at Austin, January 1983.

[TP6] R. S. Boyer and J S. Moore: "A Verification Condition Generator for FORTRAN," in The Correctness Problem in Computer Science, Robert S. Boyer and J. Strother Moore, Eds., Academic Press, London, 1981.

[TP7] R. S. Boyer and J S. Moore: "Proof Checking the RSA Public Key Encryption Algorithm," Technical Report ICSA-CMP-33, The University of Texas at Austin, 1982.

[TP8] R. S. Boyer and J S. Moore: "A Mechanical Proof of the Unsolvability of the Halting Problem," Technical Report ICSA-CMP-28, The University of Texas at Austin, January 1982.

[TP9] R. S. Boyer and J S. Moore: "Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures," in The Correctness Problem in Computer Science, Robert S. Boyer and J Strother Moore, Eds., Academic Press, London, 1981.

[TP10] Craigen, D. and M. Saaltink: "An EVES Update," ACM Software Engineering News, Vol. 10, No. 4, Aug. 1985.

[TP11] Dahl, O. and O. Owe: "A Presentation of the Specification and Verification Project 'ABEL'," ACM Software Engineering News, Vol. 10, No. 4, Aug. 1985.

[TP12] Gerhart, S. L.: "Prolog Technology as a Basis for Verification Systems," ACM Software Engineering News, Vol. 10, No. 4, Aug. 1985.

[TP13] Gerhart, S. L.: "Reusability Lessons From Verification Technology," Wang Institute of Graduate Studies Technical Report, Dec. 1983, Technical Report TR-83-04.

[TP14] Ghezzi, C. and M. Jazayeri: Programming Language Concepts, John Wiley and Sons, Inc., 1982, 1987.

[TP15] Hoare, C. A. R.: "Communicating Sequential Processes," Communications of the ACM, Vol. 21, No. 8, Aug. 1978.

[TP16] Hoare, C. A. R. et al.: "Laws of Programming," Communications of the ACM, Vol. 30, No. 8, Aug. 1987.

[TP17] Marcus, L., S. D. Crocker and J.R. Landauer: "SDVS: A System for Verifying Microcode Correctness," ACM Software Engineering News, Vol. 10, No. 4, Aug. 1985, pp. 7-14.

[TP18] Melliar-Smith, M. and J. Rushby: "The Enhanced HDM System for Specification and Verification," ACM Software Engineering News, Vol. 10, No. 4, Aug. 1985.

[TP19] Meyer, B.: "On Formalism in Specification," IEEE SOFT., Jan. 1985.

[TP20] Musser, D. R.: "Aids to Hierarchial Specification Structuring and Reusing Theorems in Affirm-85," ACM Software Engineering News, Vol. 10, No. 4, Aug. 1985.

[TP21] Musser, D. R.: "Abstract Data Type Specification in the AFFIRM System," IEEE Transactions on Software, Vol. SE-6, No. 1, Jan. 1980.

[TP22] Nyberg, K. A., A. A. Hook and J. F. Kramer: "The Status of Verification Technology for the Ada Language," IDA Paper P-1859, Institute for Defense Analyses, July 1985.

[TP23] Putnam, D.: "The VERUS Design Verification System," ACM Software Engineering News, Vol. 10, No. 4, Aug. 1985.

[TP24] N. Shankar, "Checking the proof of Godel's incompleteness theorem," Technical Report, Institute for Computing Science, University of Texas at Austin, 1986.

[TP25] Smith, M. K. and R. M. Cohen: "Gypsy Verification Environment: Status," ACM Software Engineering News, Vol. 10, No. 4, Aug. 1985.

[TP26] Williams, J. and C. Applebaum: "The Practical Verification System Project," ACM Software Engineering News, Vol. 10, No. 4, Aug. 1985, pp. 44–47.