A COMPARATIVE STUDY

OF

PRE/POSTCONDITION AND RELATIONAL APPROACHES

TO PROGRAM DEVELOPMENT

By

HONG DUAN, B.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

Abstract

With so many software-related failures happening these days, there is an increasing demand for software quality. Rigorous development approaches, which apply mathematical techniques to the design and implementation, should be getting more consideration as one of the solutions to software reliability.

Pre/postcondition approaches and relational approaches are two groups of influential rigorous techniques. Both of them use classical mathematical concepts to describe and simplify programming objects. To further propel the application of these approaches, their relative strengths and limitations in terms of practicability and accessibility need to be identified and elaborated.

In this thesis, we conduct a comparative study between the pre/postcondition approaches, proposed by Floyd, Hoare, Dijkstra and Baber, and the relational approaches, proposed by Mills and Parnas. We investigate aspects related to their mathematical models. Their abilities of specifying different termination behaviours, dealing with non-determinism, distinguishing between specifications and descriptions, etc. are discussed. Some practical issues, such as considerations on common programming constructs, side effects, verification procedures, etc. are reviewed. The comparison criteria are grouped into two categories – theory and practice. Under each criterion, we illustrate and evaluate the strength or weakness of each approach. Suggestions regarding the applications of these approaches are also presented.

Acknowledgements

I would like to express my deep and sincere gratitude to my supervisors, Dr. Robert Laurence Baber and Dr. David Lorge Parnas, for their invaluable guidance, help and encouragement throughout my graduate study and my thesis-writing period. Without their consistent support, it would not have been possible for me to finish this work.

I am grateful to Dr. Soltys and Dr. Khedri, for reviewing my thesis, and for their valuable suggestions and comments. I would like to thank my many student colleagues, especially the friends in Room 223, for providing a stimulating and fun environment and for their wonderful friendship.

Special thanks to my parents who always believe in me and have encouraged me in so many ways.

Lastly, and most importantly, I wish to thank my loving husband Yuxin for his everlasting support he has given me at the time I needed the most. He has been a source of strength in helping me maintain my sanity throughout the whole process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Mathematics is part of the everyday toolset of every working engineer. It is such an integral part of engineering that many engineers often are unconscious of the fact that they are using mathematics in their work [Par96, Par98]. However, mathematical methods have not been fully utilized in software development. Although there was an early recognition of mathematical ideas in computing, the approach of most software developers does not use mathematics. "Software development is not yet an engineering discipline, at least not in the sense commonly accepted by engineers in traditional engineering fields" [Bab97].

Software produced by standard software engineering practice typically contained around 1-3 errors per 1000 lines of code at the time of early 1990's [GY91]. At this rate, the potential for failure of software has increased as software has grown in size and complexity. According to a study conducted in 2002 by the Research Triangle Institute for the National Institute of Standards and Technology (NIST), software bugs are costing the U.S. economy an estimated $59.5 billion a year, or approximately 0.6 percent of the annual gross domestic product [RTI02]. Improving software reliability has become a rising issue. To attain high quality in software products, the standard approach is to test it. The more testing developers do of a system, the more convinced they might be of its correctness. However, testing can hardly ensure the correctness of the system in realistic software development practice. Its primary contribution to quality is to identify problems that could have been prevented in the first place.

In order to prevent errors in the early stages of development, many researchers have put a large amount of effort into developing an effective method to utilize mathematics in software development. Formal program development is the result of this process. In this area, there are several major methods. One is pre/post-condition approaches introduced by Floyd [Flo67] and further elaborated by Hoare [Hoa69] and Dijkstra [Dij75, Dij76]. Functional/relational approaches were originated by N. G. de Bruijn [deB50], Albert Meyer [MR67], et al. through their research on function theory. It is further popularized in verification and inspection by Mills [Mil75a] and Parnas [Par83] et al. Model-based approaches are also influential in formal development area. They are represented by VDM, Z and B-method.

1.2 Motivation for this research

Mathematical development techniques can provide high assurance of correctness by eliminating ambiguity and inconsistency in the early stages of development. They give precise definitions of problems. They can ensure the implementation conform to the specification. They also facilitate maintenance and reuse at the specification level instead of at the source code level. Although it is widely accepted that high-quality software would result if a rigorous development process is followed carefully, mathematical development techniques have not been applied widely in industrial projects. With so many software-related failures happening these days, formal development techniques deserve constant advocacy to popularize their application.

Why don't most software developers use those approaches? How can we facilitate the adoption of formal techniques in the software industry? To answer these questions one has to begin with the reasons behind the phenomena.

One of the reasons for the unpopularity of mathematics methods in the software industry is that practitioners traditionally have limited exposure to using mathematics in development; lack of familiarity intimidates them and keeps them from applying these methods.

Another reason is that the introduction to each approach usually concentrates on the merits of a particular approach. However a practitioner also needs to know:

1) what are the weak points of the approach and if they can be overcome, and
2) whether the approach best fits the intended application, before devoting time and effort to study how to use a particular method.

One more reason is that mathematical methods offered to the working software developer are perceived to be impractical. While they are sound mathematically, they are believed to be too difficult to use. A rigorous process is considered to be burdensome in practice, and hence is eschewed by many software developers [Par86]. It would be a mistake for researchers to ignore this sentiment and simply admonish programmers to “get with it”.

While many researchers are trying different ways to popularize various formal techniques, this thesis presents a “bird’s eye view” of several influential methods through a comparison. By going through a set of properly selected comparison criteria, we will provide a clear illustration for each approach, which will help understanding greatly.

The comparative study will offer a fresh outlook on how different formalization techniques, specification structures and verification strategies affect the applicability of an approach. The relative strengths or weaknesses of each approach in terms of its practicability can be revealed. Examples and recommendations are also provided. We believe that the discussion in this thesis presents concrete ideas and guides on how effective the approaches are when they are applied in the real world.

1.3 Thesis scope

In this thesis, we compare the pre/postcondition approaches, proposed by Floyd, Hoare, Dijkstra and Baber, with the relational approaches, proposed by Mills and Parnas. Comparisons are also conducted within each group.

Because we attempt to identify the relative strengths and limitations of an approach in terms of its practicability, the comparison criteria are chosen mostly from a user's perspective. The criteria are grouped into two categories – the theoretical aspect, which focus on the mathematical model of an approach, and the practical aspect, which focus on whether an approach provides concrete solutions to practical issues that are often encountered in development. Reflecting on the results of the comparison, suggestions and recommendations are also provided.

These formal approaches can be applied to various programs in software development. The programs we discussed in this thesis are sequential programs. They are usually considered as components of a larger program or system. The special problems involved in the development of concurrent systems will not be discussed here although some approaches could be applied in that context.

1.4 Organization of this thesis

Chapter 2 presents a historical survey of the formal program development field. Chapter 3 compares the approaches in their theoretical aspects. The individual features are described first and the differences are discussed. Chapter 4 compares the approaches in their practical aspects. Chapter 5 presents some thoughts and conclusions drawn from the comparison and gives suggestions for future work.

Chapter 2

Historical Survey of Formal Program Development Approaches

In this chapter, we present a survey of the most influential approaches in the area of formal program development.

The field of research on formal program development began in the late nineteen forties. At that time, Turing observed that reasoning about sequential programs was made simpler by annotating them with properties about program states at specific points [Tur49]. In the early sixties, John McCarthy [McC62, McC63] developed a technique called *recursion induction* for proving properties of recursively defined functions. He discussed that program correctness could be established through regarding programs as recursive functions.

There were several research issues investigated during the history of formal program development – program verification, program semantic modeling, program specification and program construction. Examples of program verification can be found as early as 1966 with Naur's work on 'general snapshots' [Nau66], which are program invariants used for correctness proofs of programs. Program verification became popular to study in the 1970s. Then, program semantics was studied on its own, that is without trying to provide an associated practical verification technique. It was popular from the late 1970s to the late 1980s [PZ01]. At about the same time, developing program specifications and deriving programs that implement the specification were also of interest to some researchers. Many approaches were developed based on different research interests. In the following sections, three major classes of approaches are described – the pre/postcondition approach, the functional/relational approach, and the model-based approach.

2.1 The pre/postcondition approach

In the late sixties, the pre/postcondition approach was proposed for proving the consistency between sequential programs and their specifications. It occurred in the work of Floyd, who first introduced the idea [Flo67], and Hoare [Hoa69]. In this approach, one generally uses a correctness proposition such as of the form: if the precondition of the program is true before the program is executed, then the postcondition will be true afterward. Floyd's and Hoare's initial work focused more on program verification. This approach is illustrated by N. Wirth in [Wir73] and by J. R. Kelley and C. L. McGowan in [MK75]. Dijkstra introduced the method known as predicate transformers [Dij76, Gri81]. A predicate transformer computes the precondition of the postcondition for a program. This computation results in an equivalent proof of program correctness. Dijkstra's approach deals with both partial and total correctness by using two kinds of transformers. Baber's approach [Bab87, Bab91, Bab02] is designed to make the traditional pre/postcondition techniques more accessible for practical applications. It provides more guidelines and procedures for applying the method in design and verification.

2.1.1 Floyd

In [Flo67], Floyd proposed a method of formally inferring the correctness of programs. The method is known as the Inductive Assertion Method. Floyd associated assertions with some points in a program. The assertions are logical formulas that characterize the set of data states that can occur when control passes to that point in the program. One proves by induction that if the execution reaches a command in a specified state that is characterized by the assertion then it reaches the next command in a specified state.

Floyd used flow charts to explain his ideas. The basic scheme is: Suppose two edges have associated assertions A1 and A2 such that there is an execution path from the first to the second. An edge is said to be an entrance/exit to the command at a vertex if its destination/origin is that vertex. An execution path is restricted to one statement or block of statements. The verification condition VC associated with these edges is then the statement that, if A1 holds when control is at its edge, and the execution path is traversed to the other edge, then A2 holds at its edge. At this point, the partial correctness of that part of the program is established. Note that this did not show that control would actually reach the second point. It only showed what would be true if it reached that point. This is called partial correctness.

Floyd also shows how to prove termination in order to establish what is called total correctness. He defined consistency and completeness for verification condition functions (*VC's*) and gave theorems for manipulating them.

2.1.2 Hoare

In [Hoa69], Hoare introduced an axiomatic approach, which was derived from Floyd's approach. He defined an axiomatic system for proving that a program is partially correct with respect to a specification. This system does not include rules for proving that a program terminates, hence it only ensures partial correctness.

Floyd's and Hoare's methods are different: Floyd verified a program through a general idea of generating and proving verification conditions of the program. Verification conditions are constructed based on the same strategy that was explained in 2.1.1. Floyd used flow charts to present general ideas of proofs using his approach. The approach can also be applied to structured/unstructured program constructs. Hoare associated Floyd's idea with the structured programming language. He used a very simple language that has only assignment, sequencing of statements, if-then-else, and while. He then provided "axiomatic semantics" for this language. The reasoning is based on an axiom (the assignment rule) together with three inference rules (the rules for consequence, composition and iteration). All the rules use some form of the expression {V}S{P}, which reads "if the precondition V is true before execution of S, then the postcondition P will be true after execution of S, provided S terminates."

2.1.3 Dijkstra

Dijkstra [Dij75, Dij76] built on the approaches of Floyd and Hoare by using a *predicate transformer* that transforms a postcondition into a precondition.

In Hoare's approach, suppose we have shown that {V}S{P} is true when V states that the initial value of the program variable is any positive integer. If we were to suspect that program S is also applicable to negative integers, we would have to repeat the proof with a modified V. Dijkstra noted that the program proof should define the set of all input values for which the output satisfies P. He provided a means of reasoning backwards from the postcondition to establish the precondition. The predicate that characterizes *all* initial states for which the program S is guaranteed to terminate in a state satisfying the postcondition P is called the weakest precondition of P for S. A weakest precondition, denoted by wp (*S*, *P*), is the least restrictive precondition that can guarantee the postcondition. It ensures the total correctness of a program. For a complete description of non-deterministic programs, another predicate transformer known as a weakest liberal precondition "wlp" is needed. *wlp*(*S*, *P*) [Dij76] is the condition that characterizes the set of *all* initial states such that activation of S will either not terminate or terminate in a state satisfying the condition P. It corresponds to partial correctness.

In [Dij75, Dij76], Dijkstra also proposed constructing programs by simultaneously deriving and verifying them from the postcondition rather than the "verification afterward" orientation of Floyd and of Hoare.

2.1.4 Baber

Unlike the authors above, Baber was developing software and discovered several practical problems that the others had overlooked or ignored. His innovations dealt with complex data structures, parameter passing, and issues that arise if a program does not always terminate [personal communication from Dr. David L. Parnas]. Baber introduced different types of preconditions – Ordinary, Strict, Complete and Semistrict, which can specify the behaviors of different programs completely [Bab87, Bab91, Bab02]. In Baber's framework, both partial correctness and total correctness of the program can be established formally. His definition of total correctness is different from others' in that total correctness of the program means the program not only terminates but also yields a defined result, i.e. a result without run-time errors. The others had considered termination because of run-time errors as a termination state like any other. Baber knew that these require special treatment in practice.

Baber paid more attention to tackling the problems which are encountered frequently in practice. He defined a concrete structure for the program state, which enables one to handle more practical issues – 1) to model the situation that run-time errors may occur, 2) to handle recursion, 3) to deal with more than one declaration of variables with the same name. Baber's model provides formal definitions for *declare* and *release* statements, which are ignored in other approaches.

To increase the applicability and accessibility of the approach, verification rules are developed in more detail, which can be used to facilitate the design. An approach combining formal and informal proof techniques in order to enable the widespread use of correctness proofs in practice is also introduced [Bab87, Bab91].

2.2 The functional/relational approach

The research of functional approaches was initiated through the study on function theory by N. G. de Bruijn [deB50] and Albert Meyer [MR67] et al. Mills [Mil75a, LMW79] then used functional semantics to provide a description of structured programs and provided a method for verifying that a program is correct with respect to an abstract specification function.

Mills' approach has no provision for non-determinacy. Parnas extended Mills' work to allow for non-deterministic programs. In Parnas' Limited Domain relation (LD-relation), he supplements the relation with a set, called the competence set [Par83], containing the states in which termination is guaranteed. In this framework, one can provide complete descriptions of both deterministic and non-deterministic programs.

2.2.1 Mills

Mills used the notion of a program as a function between initial states and final states. It is based on the observation that any deterministic program can be described by mathematical function on a program's states. "For each initial data state X for which execution terminates, a final data state Y is determined. The value Y is unique, given X, so that the set of all ordered pairs {(X, Y)} so defined is a function. We call this function the program function of a program."[LMW79].

Mills' approach can be illustrated by using his box notation "[ ]" [MBG87], which means that if *p* is a program, then [*p*] denotes the function that the program produces. If *f* is the specification of this program, then verification means showing the truth of *f* = [*p*]. However, this restricts specifications to be deterministic.

2.2.2 Parnas

Parnas extended Mills' work to allow for non-determinacy [Par83, Par97]. The behavior of a program can be described by a set of ordered pairs (x, y) where x and y are possible states of the data, and it is possible for the program to stop in state y after being started in state x. However, two non-deterministic programs with the same relational description can have different behaviour. For example, consider P1, which will always terminate in state y when started in state x, and P2, which does not always terminate when started in state x, but if it does terminate it will stop in state y. (x, y) would be part of the relation describing both programs. The competence set in LD-relations can remove this ambiguity by containing the set of states in which termination is certain. The LD-relations, introduced in [Par83, Par97], can be described by two predicates on the values of the program variables. One defines the relation between initial and final states, the other characterizes a set – the competence set – for which program termination is guaranteed. With both of them, the complete description of an unrestricted class of programs can be provided.

Parnas' research is oriented to practice and covers various issues in software engineering. In order to popularize the usage of mathematical methods in program development among practitioners, he advocates that mathematically precise documentation is of value even if proof of correctness is not attempted [Par97]. He also promotes research on inspections, which combines formal and informal techniques. To make mathematical documentation more readable, he defined tabular notations to increase the readability and verifiability of the documentation [Par92]. Tabular notations are multi-dimensional tables used to represent relations. They are particularly suitable for describing the conditions, relations and functions, which frequently occur in program documentation.

2.3 The model-based approach

Model-Based approaches, represented by VDM, Z and B-method, are methods that model the states of a system together with operations which change the state [Jon80, Spi89, Abr96]. Advocates of these methods tend to provide models of systems rather than specifications of program behaviours. A model is a simplified version of a product, while a specification is a statement of some of the properties required of a product [Par97]. In these approaches mathematical entities such as sets and functions are used to formulate a model of the system. System operations are specified by defining their effect on the system state. These approaches are based heavily on formal specification languages, which consist of a large set of specific notations and conventions. Although their underlying semantic principles differ subtly, they are similar enough to be considered as sister methods [HJN93].

2.3.1 VDM

VDM is short for Vienna Development Method, a reference to the IBM Vienna Laboratory, which started relevant work in the early 1960’s under the direction of Heinz Zemanek [Luc87]. The aim of this research group was to specify the semantics of the PL/1 programming language. The Meta-language in which the definition was written is known as VDL (short for Vienna Definition Language). In 1976, IBM decided to divert the Vienna Laboratory to other activities. The work on VDM was continued mostly by Dines Bjørner at the Technical University of Denmark, and by C. B. Jones at the University of Manchester.

What is VDM? There is no easy way to precisely answer the question. Ever since VDM emerged in the seventies, several individuals and groups have modified, redefined, extended the original “VDM”. There are basically four schools of the VDM – the Danish school, the English school, the Irish school and the Polish school. The Danish and English schools have more influence than the other two.

The Danish school has preferred the explicit operational approach [Bjo82]. Its notation tends towards the symbolic. Later the Danish school moved from VDM to RAISE [PBD85]. RAISE stands for Rigorous Approach to Industrial Software Engineering. RAISE is derived from VDM with added facilities such as concurrency, axiomatic and imperative styles. Modularity is also an integrated feature of RAISE whereas it is not supported as part of standard VDM. Some attempts have been made to add modularization to VDM but they have not yet been universally accepted.

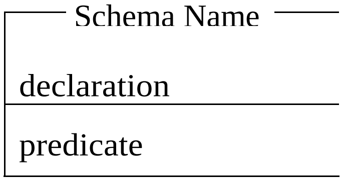
The English school uses pre/postconditions in the specification [Jon80, Jon86]. Its specifications are built in terms of models and operations specified by pre/postconditions. Steps of design by data reification or operation decomposition give rise to proof obligations. Data reification is the process of replacing an abstract mathematical data type by a more concrete representation which usually has a counterpart in the intended implementation language. The underlying logic of VDM is a non-standard logic – the three-value logic, in order to deal with partial functions. The notation adopted by the English school tends to be more verbose. Currently most researchers consider the version of the English school as VDM.

The Irish school is characterized by its constructive approach, classical mathematical style, and its notation [Air91]. In particular, this method combines the "what" and the "how" of the operation in the specification. The Polish school of VDM pursued the similar formal foundation and support tools for the VDM as RAISE through the MetaSoft project [Bli87, Bli88, Bli90].

2.3.2 Z

Z is the result of an effort started by Jean-Raymond Abrial around 1975. He continued it during his stay in Oxford. At that time the ideas of Z were taken over by the Oxford group [Spi89, PST91, Lig91, Jac97].

Z is based on concepts and notations of first-order logic and Zermelo-Fraenkel set theory. All the specifications are written in terms of pre-defined set manipulation operations. The basic specification unit in Z, called a Schema, is normally represented as:



A schema has a name, which can be used in other schemas, a declaration (also called a signature) of a set of variables (also called observations) together with their type information, and a predicate asserting the relations among these variables. A specification in Z is a collection of schemas where each schema introduces some specification entities and sets out relationships between these entities.

Z is more a notation than a method. In [Jac97], the author stated, “Z dictates few assumptions about what can be modeled…The meaning of a Z text is determined by its authors”.

2.3.3 B-method

The B-method is a method designed to be used in the specification and code generation phases of software development [Abr96, Sch01]. It was originally developed by Jean-Raymond Abrial. Later, a company called B-Core [Bco02] was formed to commercialize the method and the original B-toolkit. Also, another commercial toolset – Atelier-B – was developed by a French group [Ate02].

In B, specifications and codes are structured into the AMN (Abstract Machine Notation). The abstract machine is a concept that is very close to certain notions well-known in programming under the names of modules or classes. The main components of an abstract machine are an invariant and the operations. An invariant states properties over variables that must always hold. The operations are the only means to modify variables encapsulated by the abstract machine. Most operations have a precondition, which states the conditions under which executing the services makes sense (meets the postcondition). Operations are formalized by substitutions, which are constructs that determine and change the state of a machine. These substitutions correspond to what would be called statements in a programming language. Figure 2.1 shows an abstract machine. It is used in a queue ticket dispenser. The number of taken tickets, served tickets and waiting tickets are counted and stored.

```
MACHINE
    Ticket
VARIABLES
    take_t, serve_t
INVARIANT
    take_t∈ℕ ∧ take_t≥0 ∧ serve_t∈ℕ ∧ serve_t≥0 ∧ take_t ≥serve_t
INITIALIZATION
    take_t, serve_t := 0, 0
OPERATIONS
```

```
      Take    =
              PRE     true
              THEN  take_t := take_t + 1
              END;
      Serve   =
              PRE     serve_t < take_t
              THEN   serve_t := serve_t + 1
              END;
      ns ← ToBeServe  =
              PRE     serve_t ≤ take_t
              THEN   ns := take_t – serve_t
              END;
  END
```

Figure 2.1: An example of an abstract machine in the B-method

When developing software in B, one starts by a specification, then one writes none or several intermediate refinements. The last refinement is called an implementation. This implementation can call operations of other machines and this activity is called importation. Once the B implementation machines are written, the compiler generates a program in a target implementation language, for example, Ada or C.

There is other research that uses the notion of Abstract Machine. The term *Abstract Machine* was used by Dijkstra [Dij68] in the context of defining the operating system T.H.E. Research on the Abstract State Machine (ASM) started in 1984 by Gurevich [Gur84] based on Turing's thesis that every computable function is Turing computable. ASMs appear in embryo under the name of *dynamic structures*, later also called *evolving algebras* whose idea is that a computation is an evolution of the state [Gur93].

ASM is a language to describe software and hardware systems [Sch99]. Its author [Gur00] asserts that any computing system can be described at its natural level of abstraction by an appropriate ASM. The basic idea of ASM is the stepwise transformation of states by executing rules. The state is a collection of sets (called *universes*) with arbitrary functions and relations defined on them. The rules are the ASM statements. The most general form of rules to transform states is : if *Cond* then

Updates. *Cond* is an arbitrary condition. *Updates* consists of finitely many *function updates – f(t1…tn) := t*. In Figure 2.2, we re-write the example in Figure 2.1 using ASM's language.

```
var take_t   as natural number = 0
var serve_t  as natural number = 0
var ns as integer = 0
Take(tt as natural number)
        tt := tt + 1
Serve(st as natural number)
        st := st + 1
ToBeServe(tt as natural number, st as natural number) as integer
        return (tt – st)

Main ( )
        Take (take_t)
        if take_t > serve_t then Serve(serve_t)
        if take_t ≥ serve_t then ns := ToBeServe(take_t, serve_t)
```

Figure 2.2 An example of an Abstract State Machine

2.4 Summary

We have presented three groups of formal development approaches. Model-based approaches are based heavily on a particular specification language, which requires considerable effort to learn and use. In contrast, Floyd's, Hoare's, Dijkstra's, Baber's, Mills' and Parnas' methods have very few restrictions or assumptions on the representation. And Mills' approach has no assumptions about representation. Instead of paying great attention on how to model the program, pre/postcondition approaches and relational approaches focus more on dealing with various program behaviours. In the following chapters, we will further compare these two groups of methods.

Chapter 3

Theoretical Comparison

At the start of this work we identified a set of theoretical questions to be asked about each approach. These are summarized in Table 3.1. The remainder of this chapter discusses those questions.

Theoretical Comparison Criteria		Floyd's Inductive Assertion Method	Hoare's pre/post-conditions	Dijkstra's wp&wlp	Baber pre/post-conditions	Mills' program functions	Parnas' LD-relations
Termination of programs	Can the approach specify that a program must terminate? (see Section 3.1.1)	N[1]	N	Y	Y	Y	Y
	Can the approach specify that a program may but need not terminate? (see Section 3.1.2)	Y	Y	Y	Y	N	Y
	Can the approach specify that a program is not permitted to terminate? (see Section 3.1.3)	N	N	Y	Y	Y	Y
Does the approach distinguish between a specification and a description of a program? (see Section 3.2)		N	N	N	N	Y	Y
Non-determinism	Does the approach deal with non-deterministic specifications? (see Section 3.3.1)	Implicitly	Implicitly	Y	Y	N	Y
	Does the approach deal with non-deterministic programs? (see Section 3.3.2)	N	N	Y	N	N	Y
Is the structure of the state of program execution explicitly defined? If so, how is it defined? (see Section 3.4)		Implicitly	Implicitly	Implicitly	Y	N	N
Does the approach clearly distinguish between variables and their names? (see Section 3.5)		N	N	N	Y	Implicitly	Implicitly
Transformation rules / Proof rules	Are the rules explicitly stated? (see Section 3.6.1)	Y	Y	Y	Y	Y	Y
	Does the set of rules support verification for both partial correctness and total correctness? (see Section 3.6.2)	For partial correctness	For partial correctness	Y	Y	For total correctness	Y
How are the values of variables before and after execution distinguished? (see Section 3.7)		Intuitively	Intuitively	Intuitively	Explicitly	Implicitly	Explicitly

Table 3.1: Theoretical comparison of six approaches

[1] Y – Yes, N – No. All the further explanations are given in corresponding sections

3.1 Termination of programs

While a program may exhibit many kinds of behaviours, in this study we focus on its externally observable behaviours after termination. We are not discussing the behaviour of programs that do not terminate or the behaviour of terminating programs while running.

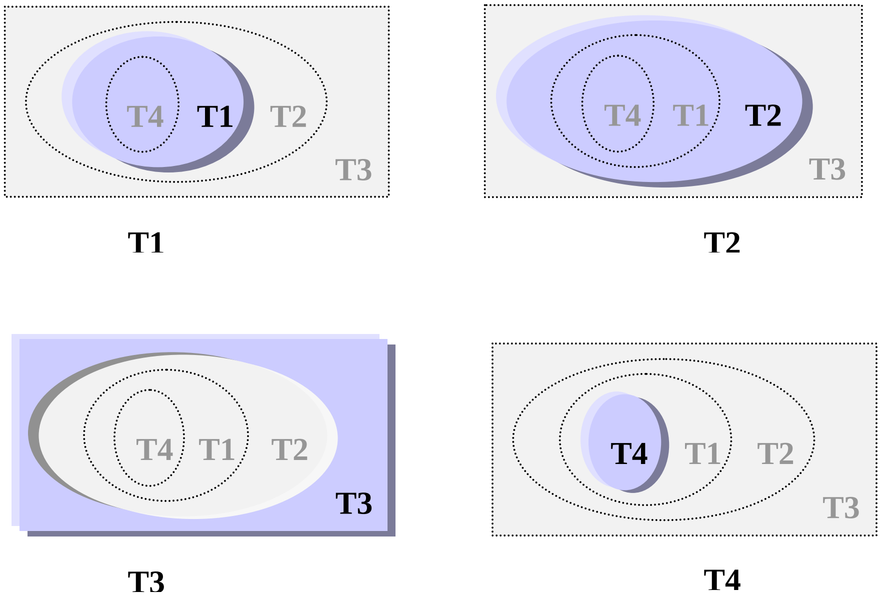
For a given starting state, the execution of the program may (a) continue indefinitely, (b) sometimes continue indefinitely and sometimes terminate, (c) always terminate in a specific state, or (d) always terminate but in one of a finite set of possible termination states. In case (a) and (c), the program is called deterministic since the result is fully determined by the starting states. In cases (b) and (d), the program is called non-deterministic. The approaches we compare treat this issue differently. We define several sets of starting states, which are illustrated in Figure 3.1, to enable further discussions on various treatments.

Definition 3.1. T1 is the set of all starting/initial states for which program P is guaranteed to terminate. T1 includes all states in cases (c) and (d).

Definition 3.2. T2 is the set of all starting/initial states for which program P may or may not terminate. T2 includes both the initial states for which termination is certain and those where termination is possible but not certain. T2 includes all states that are case (b), (c) and (d).

Definition 3.3. T3 is the set of all starting/initial states for which program P will never terminate. T3 includes all states that are case (a).

Definition 3.4. T4 is the set of all starting/initial states for which program P will terminate with all of the *intended* results computed. Note that a program may terminate after some kind of run-time error or with an error message. These final states are not the intended ones. Starting states that lead to these states are in T1 but not T4.



T1 **T2**

T3 **T4**

□ denotes the set of all possible starting states of a program.

● denotes the set under discussion.

Figure 3.1 Illustrations of four sets of starting states.

The approaches we compare may not consider all the sets defined here. They focus on different aspects of specifying the starting states of a program. Below we discuss different treatments in each approach.

Floyd, Hoare

Floyd and Hoare did not consider the four sets we defined. They [Flo67, Hoa69] introduced the precondition of a program. This precondition characterizes a set of starting states for which the program will satisfy a specific postcondition when it terminates, but the program may not terminate at all. The set described by their precondition may include the subsets of T1, T2 and T3. Their approaches cannot indicate which states are in the various sets we have identified.

Dijkstra

Dijkstra [Dij75, Dij76] defined *wp* and *wlp* to characterize particular sets of starting states. For any program S, by definition of *wp*, wp (S, true) = T1. By definition of *wlp*, wlp (S, true) = T2 and wlp (S, false) = T3.

Baber

Baber's approach is based on Hoare's approach. He defines a program/program construct as a function mapping a state to a state [Bab02, Bab87]. He does not consider non-deterministic programs. This function is typically partial, i.e. is not defined for all initial states. Baber denotes the domain of this function by $S^{-1}.\mathbb{D}$, where *S* represents the program or the program statement and $\mathbb{D}$ represents the set of all states. By definition, $S^{-1}.\mathbb{D}$ = T4. None of the others recognize the issue of intended results explicitly and define T4 accordingly.

In [HHH87], Hoare gave some consideration to an undefined expression which may cause an undefined result upon execution, "…the effect of attempting to evaluate an expression outside its domain is wholly arbitrary…". He did not provide any elaboration of that. Dijkstra took account of undefined expressions. He defined it – using the predicate D(E), which means "in the domain of the expression E". However, undefined results may be caused not only by evaluating expressions whose value is undefined, i.e. x:=1/0, but also by assigning a value to a variable which has different type. For instance, trying to assign a Boolean value to an integer variable would result in an undefined result in some languages. Baber's definitions provide a way to handle this – the value of an expression on the right-hand side of the assignment not only has to be defined, it also has to be an element of the set associated with the variable being assigned to. He provided a specific definition of domain for each major program statements and constructs.

For example, given a program statement "x:= x*5". The domain of x:= x*5 is the set of all states in which the value of the expression x*5 is an element of the set associated with the variable x, written as x*5∈Set."x". If x is a bounded variable, such as, Set.“x”= INT(0, … , 1000), then the domain of x := x*5 will be the set of states in which x*5∈INT(0, … , 1000) according to Baber’s definition. If one only considers that x*5 is defined, without considering x*5∈Set.“x”, an error of type mismatch may occur.

Mills

Mills [Mil75a, LMW79] described a deterministic program, P, by a mathematical function that is called its “program function”, written as [P]. In most of his writings, Mills did not consider non-deterministic programs. He defined that the domain of a *program function* is T1.

For the previous example, x:= x*5 where $x \in \mathbb{Z}$ and $0 \leq x \leq 1000$, in Mills’ approach, its domain will be the set $\{‘x : \mathbb{Z} \mid 0 \leq ‘x \leq 200 \wedge ‘x*5 \in \mathbb{Z}\}$ (Here we use ‘x to represent the value of x before the execution). While Mills did not explicitly identify T4, the domain of a *program function* implicitly specifies the conditions that ensure defined results. In Mills’ approach there is consideration of states in which not all of the variables of interest have defined values.

Parnas

Parnas’ approach is an extension of Mills approach in that he considers non-deterministic programs as well. In his approach the term “*program function*” is replaced by “program LD-relation” [Par83, Par97, IMP93]. He defines a LD-relation on initial and final states of the program execution, which provides a complete treatment for both deterministic and non-deterministic programs.

The definition of LD-relation:

Definition LD1. Let *U* be a set. A Limited-Domain (*LD*) relation on *U* is an ordered pair $L = (R_L, C_L)$, where:

- R_L, the relation component of *L*, is a relation on *U*, i.e., $R_L \subseteq U \times U$,
- C_L, the competence set of *L*, is a subset of the domain of R_L, i.e., $C_L \subseteq Dom(R_L)$. □

Below are the interpretations of the LD-relations with respect to the program and specification.

The definition of the LD-relation of a program P:

Definition LD2. Let *P* be a program, let *U* be a set of states, and let $L_P = (R_P, C_P)$ be an LD-relation on *U* such that:

- (x, y) $\in R_P$, if and only if *P* can terminate in state y, when started in x, and
- C_P, is the set of initial states for which *P*'s termination is guaranteed. □

The definition of what it means to satisfy a specification that is given as an LD-relation:

Definition LD3. Let $L_P = (R_P, C_P)$ be the LD-relation of a program *P*. Let *S*, called a specification, be a set of LD-relations on the same universe, and let $L_S = (R_S, C_S)$ be an element of *S*. We say that:

- *P* satisfies the LD-relation L_S, if and only if $C_S \subseteq C_P$ and $R_P \subseteq R_S$,
- *P* satisfies the specification *S*, if and only *P* satisfies at least one element of *S*. Often, *S* has only one element. □

Parnas uses the domain of a LD-relation and the competence set to specify sets of starting states. By definition, C_P = T1, Dom(R_P) = T2, and the complement set of Dom(R_P) is T3. When dealing with deterministic programs, we have Dom(R_P) = C_P. In this case we need not describe C_P and LD-relations are the same as the *program functions* in Mills' approach.

3.1.1 Can the approach specify that a program must terminate?

Floyd, Hoare

The termination will only be specified informally. As discussed previously, they cannot distinguish among T1, T2 and T3 because the sets other than T1 are not concerned in their approaches. Therefore, only partial correctness specification is provided, which ensure the program will satisfy a specific postcondition when it terminates, but the program may not terminate at all.

Dijkstra

wp(S, P) can specify those initial states for which program statement S will always terminate in a state satisfying P. For instance, we specify "n:=m-k" using *wp*:

wp ("n:=m-k", n>10) = m-k >10, which characterizes a subset of T1.

If P = TRUE, we get exactly T1.

Baber

Based on the definition of T4, Baber [Bab02] defines *strict precondition* of the given postcondition. The strict precondition V of a postcondition P specifies that the program must terminate:

$\{V\}S\{P\}\text{strictly} = \{V\}S\{P\} \wedge V \subseteq S^{-1}.\mathbb{D}$ □

The set described by a strict precondition is a subset of T4. It guarantees the termination and the intended results. We specify "n:=m-k" using a strict pre/postcondition pair,

V: { Set."n"= INT(0…1000) $\wedge$ m-k>10 $\wedge$ m-k$\in$Set."n" },

P: { Set."n"= INT(0…1000) $\wedge$ n>10 }.

If P = TRUE, V $\subseteq$ T4.

Mills

T1 is the domain of the intended function of a program. For example, in the function $f = \{(\langle a, z, x\rangle, \langle x, x!, x\rangle) \mid a\in\mathbb{Z} \wedge z\in\mathbb{Z} \wedge x\in\mathbb{Z} \wedge x\geq 0\}$, "$a\in\mathbb{Z} \wedge z\in\mathbb{Z} \wedge x\in\mathbb{Z} \wedge x\geq 0$" is used to characterize the domain of the function, i.e. T1. A program equivalent to it could be:

```
a:= 0; z:=0;
While a≠x do
    a := a+1;
```

```
        z := a*z;
    endwhile
```

Figure 3.2 An example of an implementation that must terminate

As a notational convenience, Mills [LMW79, MBG87] used concurrent assignments and conditional rules to express functions. A *concurrent assignment* summarizes the effects of several statements, mapping one state to another. A conditional rule is a sequence of (predicate → rule) pairs separated by vertical bars and enclosed in parentheses:

$$(p_1 \rightarrow r_1 \mid p_2 \rightarrow r_2 \mid \ldots \mid p_k \rightarrow r_k)$$

The meaning of this conditional rule is “evaluate predicates p_1, p_2 , …, p_k in order; for the first predicate, p_i, which evaluates True, if any, use the rule r_i; if no predicate evaluates True, the rule is undefined” [LMW79, MBG87]. Note that “$p \rightarrow r$ is not a logical implication, so we are not concerned about the truth of $p \rightarrow r$”. A concurrent assignment C can be considered as (True→C). There are also a few conventions that are used with this notational form:

(1) only the variables specified in the concurrent assignment are changed, other variables are not allowed to change,

(2) the type of variables, which is part of the domain of the function, are specified individually.

Hence, the function $f = \{(\langle a, z, x\rangle, \langle x, x!, x\rangle) \mid a \in \mathbb{Z} \wedge z \in \mathbb{Z} \wedge x \in \mathbb{Z} \wedge x \geq 0\}$, can be written as:

$$f = (x \geq 0 \rightarrow a, z := x, x!) \text{ (a an integer, z an integer, x an integer).}$$

That is, if $x \geq 0$ and a, z and x are integers, the effect of the function is described by the concurrent assignment.

Parnas

By using the competence set C_S, one can do this. Anything that can be specified using Mills’ program function can also be specified using a LD-relation. When specifying a deterministic execution, $C_S = Dom(R_S)$. When the competence set is the same as the domain of the relation, an explicit convention states that it may be omitted.

3.1.2 Can the approach specify that a program may but need not terminate?

I.e. the specification permits program to terminate but also permits it not to terminate.

Floyd, Hoare, Baber

They all defined a precondition for this kind of specification – the precondition can specify the initial states for which the execution will yield a correct result if it terminates, but it is not required to terminate.

Floyd uses the derived verification conditions based on assertions along the execution path. Hoare's triple {V}S{P} has exactly the same meaning. Baber uses the *ordinary precondition*. It is equivalent to Hoare's precondition.

Dijkstra

wlp(S, P)-wp(S, P) can specify this kind of behaviour of a program. It characterizes the set of *all* initial states for which the execution is guaranteed to yield the result satisfying postcondition P if it terminates, but it may not terminate. If P = TRUE, we get T2-T1.

Mills

Mills does not deal with this situation. With one minor exception, his papers explicitly deal only with deterministic programs so this issue does not arise. He uses a function to specify the program, which require that the program either always terminates or never terminates.

Parnas

(Dom(R_S)-C_S) specifies T2-T1 from which a program may but need not terminate.

3.1.3 Can the approach specify that a program is not permitted to terminate?

Here we mean the execution of the program results in an infinite sequence of states, but the sequence is well defined.

Floyd, Hoare

They did not take into account this kind of situation.

Dijkstra

By specifying wlp(S, P) is false, one is forbidding termination of program S in the set of states characterized by P. If P = TRUE, we get T3.

Baber

One can do this by using the *semistrict precondition* and specifying the postcondition to be false, i.e. {V} S {false} semistrictly.

Both *strict preconditions* and *semistrict preconditions* ensure that each statement in a program segment executes with a defined result. While *strict preconditions* provide extra conditions to ensure termination, semistrict preconditions do not.

Mills, Parnas

One can do this by leaving a state out of the domain of the function or relation. Programs started from states outside the domain of the function or relation will not terminate. For the example in Figure 3.2, $f = (x \geq 0 \rightarrow a, z := x, x!)$ (a an integer, x an integer, z an integer), the program started from the set of states characterized by $x<0$, which is outside the domain of the function, will not terminate.

Discussion

While most approaches can only give a specification to be either partial correctness or total correctness, a single LD-relation can specify both kinds of correctness. With R_S, one can provide the partial correctness, while with C_S, the total correctness can be specified.

3.2 Does the approach distinguish between a specification and a description of a program?

In software engineering, a precise description of the program can be required when documenting programs, and it is particularly useful in program inspections [Par94b].

As defined by Parnas in [Par94a, Par95, Par97], a *description* is "a statement of some of the actual attributes of a product, or a set of products". A *specification* is "a statement of some of the properties required of a product, or a set of product". The differences between specifications and descriptions are illustrated by the following points:

- A description must describe a product as it actually is.
- A description may include attributes that are not required.
- A specification may include attributes that a (faulty) product does not possess.
- Many visibly distinct products may satisfy a given specification.

We may use the same formalism for both of descriptions and specifications. Hence "a list of attributes should be accompanied by a statement of intent to indicate whether it is to be interpreted as a description, or as a specification. … There is never a complete description of a product" [Par97]. Here descriptions of program functional behaviour are of most interest.

Floyd, Hoare, Dijkstra, Baber

All four of these methods can be used for both description and specification but the authors did not choose to emphasize a distinction between the two activities of describing and specifying. As the example showed in Figure 3.3, the precondition-postcondition transformation describes the functional behaviour of a program, at least to the extent covered by the underlying model.

```
$\{0 \le m \wedge 0 \le n \wedge 0 \le k\}$
if $m<k$ then $m:=m+k$;
$\{0 \le k \le m \wedge 0 \le n\}$
if $n<k$ then $n:=n+k$
$\{0 \le k \le m \wedge 0 \le k \le n\}$
```

Figure 3.3: An example of a program description in pre/postcondition approaches

Mills

In Mills' approach, he used different terms when distinguishing specifications and descriptions. The intended/given function is used to specify the program. The program function is used to describe the program.

Mills [LMW79] considers two kinds of correctness of the program – *complete correctness* and *sufficient correctness*. If complete correctness is required, the domain of both the *intended function* and the *program function* are the same, i.e., Dom(*f*) = Dom([P]). If only sufficient correctness is required, the domain of the *intended function* will be a proper subset of the domain of the *program function,* i.e., Dom(*f*) $\subset$ Dom([P]), which means that the program may compute values for arguments not belonging to the domain of *f*, in other words, the program may terminate for initial states outside the domain of the *intended function*. In Mills' approach, complete correctness is always required unless otherwise specified. Therefore, the domain of the *program function* must be the same as the domain of the *intended function* to ensure desired results.

Parnas

Parnas considers and defines explicitly (Definition LD2, LD3) the distinction between descriptions and specifications. LD-relation can give precise descriptions for either deterministic programs or non-deterministic programs. Program descriptions are used primarily in documenting and inspecting programs in his approach. For the example in Figure 3.3, its description in LD-relation is:

$$\begin{aligned} R_P((‘m, ‘n, ‘k), (m’, n’, k’)) = (\ & (((‘m<‘k) \wedge (m’=‘m+‘k)) \vee ((‘m\geq‘k) \wedge (m’= ‘m))) \\ & \wedge (((‘n<‘k) \wedge (n’=‘n+‘k)) \vee ((‘n\geq‘k) \wedge (n’= ‘n))) \\ & \wedge (k’= ‘k)\ ), \end{aligned}$$

$C_P = \mathrm{Dom}(R_P)$,

where the prime before and after variables represents the initial and final values of the variable respectively.

3.3 Non-determinism

Under this criterion, we will discuss whether the approaches in question deal with non-deterministic specifications and non-deterministic programs. In 3.1, we discussed that a non-deterministic program may or may not terminate, and moreover, even the termination of the program may be also indeterminate. If a non-deterministic program terminates, it may terminate in several different states.

3.3.1 Does the approach deal with non-deterministic specifications?

Non-deterministic specifications allow any of several outcomes or non-termination for a given starting state. It states some useful properties of the result that is returned by the program, but does not fully determine what that result should be. Implementers have the freedom to change their implementation to return different results as long as the new implementation still satisfies the specification.

Floyd, Hoare,

In their systems, non-determinism is not included intentionally. It is seldom to write specifications in a non-deterministic way using their approaches.

Baber

Non-deterministic specifications are allowed intentionally in Baber's model. One can do so as follows:

V: { $0<n$ }
P: { $1\leq k \leq n+1$ }

The postcondition asks for a range of accepted values instead of requiring a specific one. It is up to the programmer to determine the implementation as long as the values of n and k satisfy the postcondition. More examples can be found in [Bab87].

Dijkstra

Non-determinism is one of the features in Dijkstra's system. *wlp* specifies the non-deterministic behaviour of a program.

Mills

In his approach, specifications are constructed using functions, which are deterministic. In [MBG87, Mil88], Mills mentioned that a program specification is a mathematical function or relation and considered the specification to be non-deterministic. However, he did not further define or formalize this consideration in his approach.

Parnas

Non-deterministic issues are fully considered and treated in Parnas' system. LD-relation can be used to construct non-deterministic specifications as defined in Definition LD3 (see 3.1).

3.3.2 Does the approach deal with non-deterministic programs?

When non-determinism happens in execution of concurrent programs, its occurrence is mostly implicit in concurrent programming since there is no explicit programming construct for expressing it. There are also sequential non-deterministic programs.

Floyd, Hoare, Baber and Mills do not discuss non-deterministic programs. Some of their ideas can apply but one must approach them with caution as they were not developed for this purpose.

Dijkstra, Parnas

Dijkstra defined a non-deterministic language – guarded command language [Dij75, Dij76]. Parnas defines a non-deterministic construct – a generalized control structure [Par83].

Dijkstra used *wp* to give meanings for non-deterministic constructs *do…od* and *if…fi* [Dij76] of his guarded command language, however *wp* alone can not, by itself, completely describe them. To provide complete semantics, one needs to use *wlp*. Dijkstra introduced *wlp* informally and did not use it to provide complete semantics for his non-deterministic constructs. From other authors [Maj80], we know that *wlp* can give the same semantics for a non-deterministic program as relational approach does.

Parnas provides complete semantics for a non-deterministic control structure using LD-relations [Par83], by which one can tell if the termination is certain or only possible or impossible.

From the definitions of *wlp* and Definition LD3, we know that *wlp* can provide the same semantics as R_P does. Also, both *wp* and C_P can specify states that ensure the termination. We have two programs P1 and P2 written in Dijkstra's non-deterministic constructs *do…od* and *if…fi* [Dij76]. P1 is a non-deterministic program and P2 is a deterministic one. They have identical behaviours when they terminate.

```
P1:   if  $x \geq 1 \rightarrow x := y$
      ▯   $x \leq 1 \rightarrow$ do $x=1 \rightarrow$ skip od
      fi
```

```
P2:   if  $x \geq 1 \rightarrow x := y$
      ▯   $x < 1 \rightarrow$ skip
      fi
```

Their descriptions using LD-relation will be

P1: $R_{P1}((`x, `y), (x', y'))$
$= (((`x \geq 1) \wedge (x' = `y) \wedge (y' = `y)) \vee ((`x < 1) \wedge (x' = `x) \wedge (y' = `y)))$,
$C_{P1} = (`x \neq 1)$.

P2: R_{P2} is the same as R_{P1},
$C_{P2} = Dom(R_{P2})$.

Using wlp and wp, we have

P1: wlp1 $= ((x \geq 1) \Rightarrow wlp\ (x:=y, P)) \wedge ((x \leq 1) \Rightarrow wlp\ (\text{do } x=1 \rightarrow \text{skip od}, P))$
$= ((x \geq 1) \wedge$ [2]$P^{x}_{y}) \vee ((x<1) \wedge P)$,

"x=1" should be excluded from the domain because the program may not terminate when x=1, thus

wp1 = (last line of wlp1 above) $\wedge$ $(x \neq 1)$.

P2: wlp2 $= ((x \geq 1) \Rightarrow wlp\ (x:=y, P)) \wedge ((x<1) \Rightarrow wlp\ (skip, P))$
$= ((x \geq 1) \wedge P^{x}_{y}) \vee ((x<1) \wedge P)$,

wp2 $= ((x \geq 1) \Rightarrow wp\ (x:=y, P)) \wedge ((x<1) \Rightarrow wp\ (skip, P))$
$= ((x \geq 1) \Rightarrow P^{x}_{y}) \wedge ((x<1) \Rightarrow P)$
= wlp2.

In this example, *wlp* and *wp* together give the same descriptions for both deterministic and non-deterministic programs as the LD-relation does.

[2] P^{x}_{E} is the predicate we get by replacing all occurrences of x in *P* with E [HW73]. It is often used in pre/postcondition approaches.

3.4 Is the structure of the state of program execution explicitly defined? If so, how is it defined?

There are two distinct approaches to defining or describing the state of execution of a program. The most common one views the data state as a collection of variables with (at least) names and values. Mills, Parnas, and others make no such assumption. They express their semantics in terms of states without saying how they are represented. Of course, to apply the methods one will need to chose a representation and this is likely to be the same as the representation used in one of the other approaches.

Floyd, Hoare, Dijkstra

They implicitly assume that states are described in terms of a mapping from variable names to values, for example {(pi, 3.1415926), (dd, “day”), (y, 66)}.

Baber

Only Baber’s approach explicitly defines the structure of the state of program execution. A *data environment* is defined to represent a state. It is a sequence of program variables, for example: [(x, N, 1), (y, R, 5.77), (z, Strings, “abc”), (x, N, 10)]. Each program variable is defined as a triple, consisting of a name, a set and a value. The value must be an element of the set. Although in Baber approach, there are more restrictions on the representation comparing to Floyd et al’s approaches, Baber representation is more capable of representing the distinctions between the states.

By defining the structure in this way, Baber can facilitate his notion of “the value assigned to the variable must be an element of the set associated with the variable”. One must ensure that each variable satisfies the definition and hence the resulting sequence of triples would be a data environment. For instance, trying to execute the assignment statement x:= 5 with a data environment in which the variable named x is associated with the set of Boolean values would result in setting the variable x to the triple (x, Boolean, 5) which is not a program variable (as defined above), so this operation will lead to an undefined result.

Mills, Parnas

In their framework, one does not assume any particular state representation.

Discussion

Different design goals give rise to the difference on state representations. A general view of states, like Mills and Parnas, will provide flexibility when an approach is applied in different contexts. The general views also make the approaches simpler to apply because they do not have rules on how to manipulate the representations. On the other hand, they give less guidance. A programmer has to decide what kind of state representation s/he should adopt. The structure in Floyd et al.'s approaches is assumed as a function between variable names and their values, whereas in real program, this assumption is not valid. Baber uses a more concrete representation. This model is appropriate for the vast majority of programs. With this concrete structure, Baber's model can distinguish between certain "run-time errors" and "normal" outcomes of executing a program. In addition, Baber's particular state representation gives a didactical advantage to the approach. Especially for software developers in practice, a concrete view can give ease of understanding. Mills and Parnas could introduce the same information as Baber does when they pick a representation for an actual application. Their approach differs from all the others in that the assumption is not built into the method but chosen by the user.

3.5 Does the approach clearly distinguish between variables and their names?

Variables are the basic units for storing data, i.e. they can be thought of as a sort of "containers". The value of the variable is the content of such a container. At any moment they will have a value. Each variable has a name, which can be used to refer to its value and a type, which determines what values the variable can hold and what operations can be performed.

During a variable's lifetime, the binding between this variable and its name can change dynamically during the program execution. Their relationship is not necessarily one-to-one correspondence. A variable may have one or more names – aliasing. This happens either during parameter passing or in complex data structure (array, structure, etc.). Sometime variables may share the same name (e.g. because of block structure rules). A clear distinction between the variables and their names is necessary to deal with various binding situations in verification.

Floyd, Hoare, Dijkstra

In these approaches, they make simplifying assumptions that all the variables have unique names and that no variable has two names, e.g. a one-to-one correspondence between variables and their names. Variables and their names are not distinguished during the formalization. Due to this assumption, the possibility of having two variables with the same identifier is ignored. And these approaches cannot deal with the possibility that a variable will have two names at some point in the program (aliasing). It makes the approach being inapplicable for many real programs.

Suppose we have a simple pre/postcondition formula {a=1}b:= a+2{a=1}. If a and b are refer to the same variable, i.e. a and b are aliased, then the formula will be invalid. The problem is ignored in their approaches instead of handling it.

Baber

Baber does not make this assumption. His definition of variables, as we have seen in 3.4, clearly distinguish the variable and its name. It adds much more flexibility to the traditional approach. The value of the variable *x* is defined to be the value of the first program variable in a data environment whose name is *x*. It is possible that a data environment may contain more than one program variable with the same name. Even identical program variables may appear in a data environment. Baber also introduced *hidden variables* [Bab02], which allow the user to relate the pre/postcondition for an individual declare and release statement. This variable also add capability for handling recursion.

Mills, Parnas

They avoid this issue by not specifyng a representation. The representation chosen could be Baber's if they want to handle the problems being discussed here.

Discussion

To distinguish the variable and its name is a practical issue in applying the approach to real program. Things like references/pointers are all related to it. Since the introduction of pointers into high-level languages, there have been warnings against them. For example, Hoare criticized the introduction of pointers into high-level languages [Hoa74]. Formal methods advocates see references, especially aliasing, as the primary obstacle in verification [Kul02]. However, practitioners consider it as a necessary tool for efficient implementation. Some researchers have simply ignored these problems by telling programmers to avoid them. Instead of ignoring, Baber has shown how to deal with them.

3.6 Transformation rules / Proof rules

The rules we discuss here are the set of procedures or standards to allow proofs or justify transformations of expressions to equivalent or stronger/weaker expressions.

3.6.1 Are the rules explicitly stated?

All these approaches provide certain rules to facilitate verification or design.

Floyd, Hoare, Dijkstra, Baber

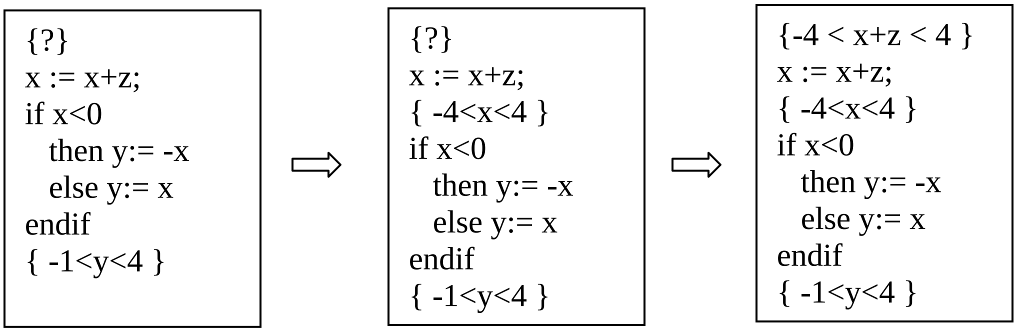
In pre/postcondition approaches, all provide a set of systematic rules for transforming predicates on states based on the program statements (see Section 4.1).

In pre/postcondition approaches, one decomposes and reduces the theorems (given as specifications) based on the transformation rules associated with program statements/constructs to prove the correctness of a program. Programs are put into a context of a logic system. Transformation rules are, actually inference rules, given for a small set of statement types. One applies rules repeatedly throughout the program.

For example, in Figure 3.4, a program segment and a postcondition are given. A precondition of this segment needs to be derived in order to prove its correctness. First, according to the transformation rules for the if and assignment statements in pre/postcondition approaches, we have a formula for the precondition of the if statement:

$$((-1<y<4)^{y}_{-x} \wedge x<0) \vee ((-1<y<4)^{y}_{x} \wedge x\geq 0)$$

which can be reduced to -4<x<4. Then, according to the proof rule for the assignment statement, we have $(-4<x<4)^{x}_{x+z}$ as the precondition of the whole segment. It can be reduced to -4<x+z<4.



```
{?}
x := x+z;
if x<0
  then y:= -x
  else y:= x
endif
{ -1<y<4 }
```

```
{?}
x := x+z;
{ -4<x<4 }
if x<0
  then y:= -x
  else y:= x
endif
{ -1<y<4 }
```

```
{-4 < x+z < 4 }
x := x+z;
{ -4<x<4 }
if x<0
  then y:= -x
  else y:= x
endif
{ -1<y<4 }
```

Figure 3.4: An example of transformation rules in pre/postcondition approaches

In the course of proving the correctness of a program, one can either derive the precondition of a program from its postcondition, or in a reverse order, derive a postcondition from a precondition. The former is used in Dijkstra's approach [Dij76] and the latter is usually done in Hoare's [Hoa69, Hoa71a].

Mills, Parnas

They provide general rules about how to construct program functions/LD-relations. These rules are actually definitions of programming language constructs. One follows the rules to generate program relations, then prove the equivalency between intended relations and program relations.

The rules given are compact and abstract, however, the authors intended not to provide a specific representation so that one has to build his own more concrete rules when using them. Mills did provide some concrete solutions in his books [LMW79, MBG87], but they are not the rules one has to stick to and do not restrict one from choosing his/her own representation.

Rules can be applied hierarchically by summarising long programs by a single relation based on the relations of its component parts. For example, the program relation of the example in Figure 3.4 can be constructed using the convention in Parnas' approach:

$$\begin{aligned} & R_P((`x, `y, `z), (x', y', z')) \\ = & R_{P1} \circ R_{P2} \\ = & \{(x' = `x + `z) \wedge (z' = `z)\} \circ \{((`x<0) \wedge (y' = -`x)) \vee ((`x \geq 0) \wedge (y' = `x))\} \end{aligned}$$

where R_{P1} and R_{P2} represent the program relations of the assignment statement and the if statement respectively. Then, by doing the relational composition, we have the relation that describes the program:

$$\begin{aligned} & R_P((`x, `y, `z), (x', y', z')) \\ = & \{((`x + `z < 0) \wedge (y' = -`x - `z) \wedge (z' = `z)) \vee ((`x + `z \geq 0) \wedge (y' = `x + `z) \wedge (z' = `z))\} \end{aligned}$$

3.6.2 Does the set of rules support verification for both partial correctness and total correctness?

Floyd, Hoare

They only provide rules for partial correctness verification. Floyd provided partial correctness rules for different command types of a flow chart language and for Algol and some special constructs, such as Go-To and LABEL. Hoare provides partial correctness rules for most of the constructs in a Pascal-like language [Hoa69, HW73, HHH87].

Dijkstra, Baber

The rules associated with *wp* support total correctness verification. From the definition of *wlp*, we know that *wlp* can specify partial correctness for a program, but Dijkstra did not provide verification rules using *wlp*.

Baber provides complete rules for both partial and total correctness verification. His rules were developed at a more practical level than Dijkstra's when dealing with total correctness.

Mills, Parnas

Mills' method deals with total correctness by applying his rules of constructing program functions. Parnas' LD-relation can be reduced to Mills' function when dealing with total correctness, and all the rules in Mills' method can also apply. LD-relation can also specify partial correctness of a program but the related verification rules are not provided.

3.7 How are the values of variables before and after execution distinguished?

To specify the before/after behaviour or input/output relation for a program, it is necessary that one be able to refer to the initial value of the variable.

	Floyd	Hoare	Dijkstra	Baber	Mills	Parnas
Initial value	No specific convention. Distinguished intuitively.			X’	No specific convention. Distinguished implicitly.	‘X
Final value				X		X’

Table 3.2 Initial/Final values of variables

Note that, the references to initial values in Baber and Parnas’ method are opposite in that a prime after means “before” in Baber’s while it means “after” in Parnas. Parnas uses the absence of primes to indicate that the value does not change.

Floyd, Hoare, Baber, Dijkstra

In “pure” pre/postcondition approaches, each assertion only characterizes the current state in terms of program variables. In the original version, Hoare [Hoa69] followed this notion and his assertion on postcondition did not have any references to initial values. However, in many situations, it is necessary to refer to the initial value of a variable in order to describe the program behaviour precisely.

For example, we have a pre/postcondition pair {x>y} and {y>x}. One can simply use an assignment to achieve the postcondition. Only if one refers to the initial values could one demand that the final values be a swap of the initial values. The most famous example was presented by Yelowitz and Gerhard [GY76]. It illustrated the necessity of referring to the initial values in the postcondition in the case of specifying a program that sorted an array. The postcondition not only needs to state that the value in the array are sorted, but also has to refer to the initial values to state that the final values are a permutation of the initial values. Without specifying the permutation, the postcondition would be satisfied by a program that simply filled array A[i] with i.

In fact, extra variables, which refer to initial values in a postcondition, have appeared in Floyd's, Hoare's and Dijkstra's work. These variables were used intuitively and were not formally introduced. Much literature on pre/postcondition approaches gave this artificial variable different names, such as auxiliary variable [Kle98], dummy variable, logic variable [Gri81] and specification variable [Kal90], etc. The notion of pre/postcondition approaches is to characterize the current state of execution, while artificial variables have nothing to do with the current state of execution. They are used to indicate a relation between initial and final states. When one adds them in the postcondition, the postcondition indeed becomes a *relation* on initial and final states.

Baber extends the traditional notion of a pre/postcondition approach. In his approach, variable names in conditions are functions mapping the state (data environment d0, d1, …) in question to a value, specifically, the value of the first variable in the data environment with the name in question. Thus, the term *x*' is not an intuitive "add-on". It is to be interpreted as x.d0 and the term *x* to be interpreted as x.d1, where d0 and d1 are the initial and final states respectively.

Mills, Parnas

In relational approaches, one refers to the value of initial and final state directly and does not have to add any extra variables. The only variables used are those that represent the values of program variables.

Mills used X_0, X_1, X_2, etc. intuitively to distinguish the initial and final values. Parnas gave explicit considerations on this issue. For the above example of swap, the program relation can be represented in Parnas' convention as R_P = ((x' = 'y) $\wedge$ (y' = 'x)). The relationship between initial and final states is clearly described.

3.8 Conclusion

Floyd’s and Hoare’s methods have limited abilities on specifying program behaviours because their concerns are mostly focus on partial correctness.

Dijkstra’s wp&wlp and Parnas’ LD-relations can handle both deterministic and non-deterministic programs. They are based on different viewpoints of formalization, but both have powerful expressive abilities that enable specifiers to deal with all three kinds of terminating behaviours of programs. In addition, these two methods are capable of specifying both partial. and total correctness.

Mills’ program functions method focuses on deterministic programs and total correctness. His and Parnas’ methods provide abstract rules that are simpler and less restrictive but also result in less guidance for the users.

Baber’s method also has strong expressive ability on specifying terminating behaviours of programs, and can handle partial and total correctness as well. Only his method recognizes the intended results of a program and can further specify them. Baber’s method is intended to provide more guidance and concrete rules to facilitate specifying and verifying programs.

Chapter 4

Practical Comparison

When we began this research, we identified a set of practical questions to be asked about each approach. These are summarized in Tables 4.1 and 4.2. The remainder of this chapter discusses those questions.

Practical Comparison Criteria					Floyd's inductive Assertion Method	Hoare's pre/post-conditions	Dijkstra's wp&wlp	Baber's pre/post-conditions	Mills' program functions	Parnas' LD-relations
Types of program statements considered	Primitive program statements (see Section 4.1.1)	Does the approach define a fixed set of primitive program statements?			Y[1]	Y	Y	Y	N	N
		Is Abort handled?			N	Y	Y	Y	Y	Y
		Is Skip/Null handled?			N	Y	Y	Y	N	N
		Are assignment statements handled?			Y	Y	Y	Y	Y	Y
		Are declaration and release statements handled?			N	Partially	N	Y	N	N
	Control constructs (see Section 4.1.2)	Is sequential composition handled?			Y	Y	Y	Y	Y	Y
		Is conditional composition handled?			Y	Y	Y	Y	Y	Y
		Are loop constructs handled?			Y	Y	Y	Y	Y	Y
		Are procedure calls handled?	A procedure call without formal parameters		N	Y	N	Y	Y	Y
			A procedure call with formal parameters	Call by value	N	Partially	N	Y	N	N
				Call by name	N	Partially	N	Y	N	N
				Others	N	N	N	Call by value-result	N	N
		Is the recursive invocation of programs handled?			N	Partially	N	Y	Y	N

Table 4.1: Practical comparison of six approaches (1)

[1] Y – Yes, N – No. All the further explanations are given in corresponding sections.

Practical Comparison Criteria	Floyd's Inductive Assertion Method	Hoare's pre/post-conditions	Dijkstra's wp&wlp	Baber's pre/post-conditions	Mills' program functions	Parnas' LD-relations
How does the approach support verification of a program? (see Section 4.2)	4.2	4.2	4.2	4.2	4.2	4.2
Side effects handled? (see Section 4.3)	N	N	N	N	Y	Y
How does the notation of an approach facilitate its use in documentation? (see Section 4.4)	Math	Math	Math	Math , natural language &diagrams	Conditional rules & trace tables	Tabular Notation
How does the approach handle program derivation / design? (see Section 4.5)	N	N	Derivation/ Stepwise refinement	Derivation/ Informal design guidelines	Stepwise refinement	4.5
Programming languages limitation (see Section 4.6)	Imperative, OOP	Imperative, OOP	Imperative, OOP	Imperative, OOP	Imperative, OOP	Imperative, OOP
Does the approach provide explicit descriptions on how to use the transformation/proof rules? (see Section 4.7)	4.7	4.7	Y	Y	Y	Y

Table 4.2: Practical comparison of six approaches (2)

4.1 Types of program statements considered

Specifications enable us to write correct programs. Verification proves programs' consistency with their specifications. However, these will be impossible if we do not know what the various statements and constructs in the programming language are supposed to do. The meaning of the program needs to be defined and understood.

We will discuss the semantics of common primitive program statements and program constructs defined in these approaches[2]. Most of these definitions are also used as rules for proving programs' correctness or assisting the program design.

4.1.1 Primitive program statements

- **Does the approach define the basic semantics for a fixed set of primitive program statements?**

  Floyd, Hoare, Baber, Dijkstra

  All define the basic semantics for some primitive program statements in order to provide a logical basis for proofs of the program correctness.

  Floyd provided the semantic definition for each common statement type, which is also the rule for constructing a *verification condition* $VC(P, Q)$ [Flo67]. Hoare [Hoa69] defines an inference system. In this system, the assignment rule serves as an *axiom* and *inference rules* are introduced based on the axiom. Dijkstra [Dij76] defined predicate transformers for program constructs, which are the "rules of inference" in his system. These predicate transformers work backwards to transform the predicate characterizing the postcondition to the one

[2] Dijkstra introduced *wlp* to specify the non-deterministic behaviour of a program. But he did not use *wlp* to provide semantic definitions for either program statements or constructs. The formulae using *wlp* that are presented in the follows are constructed only through his definition of *wlp*.

characterizing the precondition. Baber [Bab02] defines the relationships, called *proof rules* (his “rules of inference”), between preconditions and postconditions according to each statement type.

Mills, Parnas

They do not define the basic semantics for primitive program statements. The axioms in their systems would be the definitions of the basic functions/relations associated with a set of programs.

Mills’ rules of inference are more abstract than the others. They are the rules for composition of functions. Parnas followed Mills in that but allowed relations. Both of them derive the function/relation of a constructed program from the function/relation of its components. Mills [Mil75a, LMW79, MBG87] provided composition rules for control constructs of structured programs, which are built on the algebra of functions. Parnas [Par83] defines the algebra of LD-relations which are slightly different of conventional relations and functions. Parnas also defines a generalized control structure and provides its formal definition using LD-relations.

Discussion

One of the major differences between pre/postcondition approaches and relational approaches are due to the different way that they define the semantics of the program. In the pre/postcondition approaches, the meaning of a program is based on the definitions of basic program statements/constructs. In the relational approaches, the meaning of a program is based on the notion that a program can be described by a relation between the initial and final states, no matter if it is for a large program or a basic statement/construct, their semantics are captured in same way.

- **Abort**

Abort is a program that fails to terminate properly. In terms of pre/postcondition, it always fails to establish any postcondition. Some formal systems use mathematical theories that admit its existence in order to prove the absence of such an error [Dij76]. In Baber's approach, any program with an empty domain corresponds to others' *abort* statement, e.g. x:=1/0.

Floyd	No definition.
Hoare	{V} abort {False}[HHH87]
Dijkstra	wp (abort, P) = False [Dij76]
	wlp (abort, P) = True
Baber	Empty domain [Bab02]
Mills	Empty function [LMW79]
Parnas	Empty relation [Par83]

Table 4.3: Abort

- **Skip/Null**

The Skip statement is an empty statement. Its execution always terminates, leaving everything unchanged. Except the approaches in Table 4.4, other approaches did not consider it.

Hoare	{P}skip{P} [HW73]
Baber	Empty statement [Bab02]
Dijkstra	wp ("skip", P) = P [Dij76]
	wlp ("skip", P) = P

Table 4.4: Skip

- **Assignment statements**

Table 4.5 shows the definitions for assignment statements used in pre/postcondition approaches and the program functions of assignment statements in relational approaches.

Floyd	For assignment statement, x:= E, its *Verification Condition* *VC* = $(\exists x_0)\,(S^{x}_{x0}\,(V) \wedge x = S^{x}_{x0}\,(E)) \Rightarrow P$, S^{x}_{x0} means substituting x_0 for each occurrence of x and E is considered as a true function of its free variables. [Flo67]
Hoare	P^{x}_{E} {x:=E}P P^{x}_{E} is the predicate we get by replacing all occurrences of x in *P* with E. (same meaning applied in other approaches) [HW73]
Dijkstra	wp (“x:=E”, P) = D(E) **cand**[3] P^{x}_{E} [Dij76]
	wlp (“x:=E”, P) = P^{x}_{E}
Baber	Proof rules for total correctness: 1. {P^{x}_{E} and E ∈ Set.“x”} x:=E {P} completely and strictly. 2. If V ⇒ P^{x}_{E} and E ∈ Set.“x” then {V} x:=E {P} strictly. [Bab02]
	Proof rules for partial correctness: 1. { P^{x}_{E} } x:=E {P} completely. Used to derive a precondition for a given postcondition and assignment statement. 2. If V ⇒ P^{x}_{E} then {V} x:=E {P}. Used for verification. [Bab02]
Mills	[x:=E] = {((x, y, …), (E, y, …))} [LMW79]
Parnas	R_S = {(‘x, ‘y, …), (x’, y’, …)$\mid$ x’ = ‘E $\wedge$ y’ = ‘y $\wedge$ … } C_S = Dom(R_S) [Par83]

Table 4.5: Assignment statements

[3] **cand** represents *conditional conjunction* [Dij76]. A conditional conjunction is one in which the second operand is evaluated if and only if this is necessary to determine the value of the conjunction. When both operands are defined, the boolean expression “P cand Q” has the same value as “P and Q”. “P cand Q” is also defined to have the value “false” where P is “false”, regardless of the question whether Q is defined.

Discussion

- In Mills' approach, one can also write the function for [x:=E] as {((x, y, …), (u, v,…)) | u=E $\wedge$ v=y $\wedge$ … }. The variable's name in the assignment statement and the associated program function could be completely different because the purpose of the function is to reflect the changes of states caused by the assignment. On the other hand, this brings the problem that the programmer has to decide how to represent the states in a consistent way in the formulation. Mills [LMW79, MBG87] used (x:=E) to represent such functions. Parnas also introduces a concrete way (variable names with the before and after prime) for representation.

- Compared to Dijkstra's approach, Baber's specification on total correctness gives stricter conditions. The value of E not only has to be defined, it also has to be an element of the set associated with variable x. In Parnas' approach, the type of the variable are usually not specified since it is assumed that the information is provided elsewhere.

- Most programming languages allow more complicated left-hand sides of assignments, including array references and pointers. Both Mills and Baber provide a method with examples of handling the array assignment [LMW79, Bab02]. A pointer can be viewed as an index (subscript) to an array. While none of the approaches provide explicit treatment for pointers, Baber's and Mills' approaches that handle arrays could handle pointers easily.

- **Declaration and release statements**

Most approaches have no treatment of declaration and release statements. Hoare provides a rule, which is only intended to handle the local variable declaration in a subprogram. Baber's treatments on these two statements assist to avoid and detect the scope problem of local variables through giving a clear indication on the variable's lifetime. Though Baber does not consider scope rules [Bab02], the scope of a variable could be handled by his proof rules of declare and release statements. It is particular useful when dealing with subprogram calls with formal parameters and recursion problems.

Hoare	$\dfrac{\{V\}S^{x}_{y}\{P\}}{\{V\}\mathrm{new}\ x; S\{P\}}$, where y is not free in V or P, nor does it occur in S (unless y is the same identifier as x). [HW73]
Baber	Proof rules for total correctness, Declaration 1: $\{P^{x,\,Set.''x''}_{E,S}$ and $E \in S\}$ declare (x, S, E) {P} completely and strictly. $P^{x,\,Set.''x''}_{E,S}$ means replacing 1) every reference to the value of the variable x in P by E and 2) every reference to the set associated with x in P by S. Declaration 2: If $V \Rightarrow P^{x,\,Set.''x''}_{E,S}$ and $E \in S$ then {V} declare (x, S, E) {P} strictly. For array variable declaration: If $V \Rightarrow P^{x(ie),\,Set.''x(ie)''}_{E,S}$ and $E \in S$ and ie$\in$**Siv** then {V} declare (x(ie), S, E) {P} strictly. **Siv** is the set of permitted index values. Release 1: If {V} release x {P} and V $\Rightarrow$ Set.“x”$\neq\varnothing$ then {V} release x {P} strictly. Release 2: If {V} release x {P} then {V and Set.“x” $\neq\varnothing$} release x {P} strictly. [Bab02]
	Proof rules for partial correctness, Declaration 1: $\{P^{x,\,Set.''x''}_{E,S}\}$ declare (x, S, E) {P} completely, Declaration 2: If $V \Rightarrow P^{x,\,Set.''x''}_{E,S}$ then {V} declare (x, S, E) {P}. Release: {B} release x {B}, if the value of B is not affected by the execution of release x. [Bab02]

Table 4.6: Declaration and Release statements

4.1.2 Control constructs

- **Sequential composition**

A program may consist of a sequence of statements which are executed one after another. We use S1; S2 to denote it here.

Floyd	$VC\ (Vs, Ps_1, Ps_2; Vs_1, Vs_2, Ps) = (Vs \Rightarrow Vs_1) \wedge (Ps_1 \Rightarrow Vs_2) \wedge (Ps_2 \Rightarrow Ps)$ Floyd treated a sequence of statements or a group of statements in a certain construct as a *compound statement*. *Vs* and *Ps* are the pre and postcondition of this compound statement. Vs_1, Vs_2 and Ps_1, Ps_2 are the preconditions and postconditions of S1 and S2 respectively. [Flo67]
Hoare	$\dfrac{\{V\}S1\{P1\},\ \{P1\}S2\ \{P\}}{\{V\}\ S1;S2\ \{P\}}$ [HW73]
Dijkstra	wp (“S1;S2”, P) = wp (S1, wp (S2, P)). [Dij76]
	wlp (“S1;S2”, P) = wlp (S1, wlp (S2, P)).
Baber	A proof rule for total correctness: If {V} S1 {P1} strictly and {P1} S2 {P} strictly then {V} (S1, S2) {P} strictly. [Bab02]

	A proof rule for partial correctness, besides a similar rule to Hoare’s, an additional rule about a sequence of assignment statements:
	If $V \Rightarrow [[\ \ldots\ [P^{xn}_{En}]\ \ldots\ ]^{x2}_{E2}]^{x1}_{E1}$
	then {V}
	x1:=E1
	x2:=E2
	...
	xn:=En
	{P}. [Bab02]

(*continued on next page*)

Mills	The program function for sequential composition: $[g; h] = \{(X, Y) \mid Y = h \circ g(X)\}$, where X is the initial state, Y is the final state. [Mil75a, LMW79]
Parnas	$R_{LS1;S2} = R_{S1} \circ R_{S2}$ $= \{(x, y) \mid (\exists z)\ ((x, z) \in R_{S1}) \wedge ((z, y) \in R_{S2})\}$ The competence set is the same as the domain of the relation for deterministic programs, which are all that are being considered here. [Par83]

Table 4.7: Sequential composition

- **Conditional composition**

The conditional composition we discussed here is the “**if** B **then** S1 **else** S2 **endif”** commonly used in structured programming.

Floyd	$VC\ (Vs, Ps_1, Ps_2\ ; Vs_1, Vs_2, Ps) = ((Vs \wedge B) \Rightarrow Vs_1) \wedge ((Vs \wedge \neg B) \Rightarrow Vs_2) \wedge ((Ps_1 \vee Ps_2) \Rightarrow Ps)$ *Vs* and *Ps* are the pre and postcondition of this compound statement. *Vs₁*, *Vs₂* and *Ps₁*, *Ps₂* are the preconditions and postconditions of S1 and S2 respectively. [Flo67]
Hoare	$\frac{\{V \wedge B\}S1\{P\},\ \{V \wedge \neg B\}S2\ \{P\}}{\{V\}\ \text{if B then S1 else S2}\ \{P\}}$ [HW73]
Dijkstra	wp (if, P) $= D(B) \wedge ((B \wedge wp\ (S1, P)) \vee (\neg B \wedge wp\ (S2, P)))$ [Dij76]
	wlp (if, P) $= (D(B) \Rightarrow ((B \wedge wlp\ (S1, P)) \vee (\neg B \wedge wlp\ (S2, P))))$

(*continued on next page*)

Baber	Proof rules for total correctness: (For both total correctness and partial correctness, the first rule is for verification and the second is for deriving a precondition) 1. If $V \Rightarrow B \in \{false, true\}$ and {V and B} S1 {P} strictly and {V and not B} S2 {P} strictly then {V} if B then S1 else S2 endif {P} strictly. 2. If {V1} S1 {P} strictly and {V2} S2 {P} strictly then {V1 and (B=true) or V2 and (B=false)} if B then S1 else S2 endif {P} strictly. [Bab02]
	Proof rules for partial correctness: 1. If {V and B} S1 {P} and {V and not B} S2 {P} then {V} if B then S1 else S2 endif {P}. This rule is the same as Hoare's. 2. If {V1} S1 {P} and {V2} S2 {P} then {V1 and B or V2 and not B} if B then S1 else S2 endif {P}. [Bab02]
Mills	The program function for conditional construct: [if *b* then *g* else *h*] $= \{(X, Y) \mid (b(X) \rightarrow Y=g(X)) \wedge (\sim b(X) \rightarrow Y=h(X))\}$, where X is the initial state, Y is the final state. [Mil75a, LMW79]
Parnas	$R_{Lif} = R_{B\rightarrow S1} \cup R_{\neg B\rightarrow S2}$ $= \{(x, y) \mid (B(x) \wedge (x, y) \in R_{S1}) \vee (\neg B(x) \wedge (x, y) \in R_{S2}))\}$ $C_{Lif} = Dom(R_{Lif})$ [Par83]

Table 4.8: Conditional composition

Example 4.1: Verify a program segment which uses IF statement.

```
S:  if x<y
      then z := x
      else  z := y
      endif
```

➢ In pre/postcondition approaches, a specification for S is given as:

V: $\{x>0 \wedge y>0 \wedge z>0\}$
P: $\{x>0 \wedge y>0 \wedge z>0 \wedge z = \min(x, y)\}$
and where min is a function that returns the minimum of two arguments.

- In Floyd's approach, we need four more assertions along the execution path for the verification:

```
if x<y
then
            Vs1: {x>0 ∧ y>0 ∧ z>0 ∧ x<y}
z:=x
            Ps1: {x>0 ∧ y>0 ∧ z>0 ∧ z=x}
else
            Vs2: {x>0 ∧ y>0 ∧ z>0 ∧ x≥y}
z:=y
            Ps2: {x>0 ∧ y>0 ∧ z>0 ∧ z=y}
endif
```

According to his rule of conditional composition, we need to prove
$((V_S \wedge B) \Rightarrow V_{S1}) \wedge ((V_S \wedge \neg B) \Rightarrow V_{S2}) \wedge ((P_{S1} \vee P_{S2}) \Rightarrow P_S)$.

Proof:

$$
\begin{aligned}
& Vs \wedge B \\
= \; & \\
& x>0 \wedge y>0 \wedge z>0 \wedge x<y \\
= \; & \\
& Vs_1 \; \blacksquare
\end{aligned}
$$

$$
\begin{aligned}
& Vs \wedge \neg B \\
= \; & \\
& x>0 \wedge y>0 \wedge z>0 \wedge x \geq y \\
= \; & \\
& Vs_2 \; \blacksquare
\end{aligned}
$$

$$
\begin{aligned}
& Ps_1 \vee Ps_2 \\
= \; & \\
& (x>0 \wedge y>0 \wedge z>0 \wedge z=x) \vee (x>0 \wedge y>0 \wedge z>0 \wedge z=y) \\
\Rightarrow \; & \\
& x>0 \wedge y>0 \wedge z>0 \wedge z = \min(x, y) \\
= \; & \\
& Ps \; \blacksquare
\end{aligned}
$$

Thus, $((Vs \wedge B) \Rightarrow Vs_1) \wedge ((Vs \wedge \neg B) \Rightarrow Vs_2) \wedge ((Ps_1 \vee Ps_2) \Rightarrow Ps)$ holds.

- In Hoare's and Baber's approaches, we need to prove $\{V \wedge B\}S1\{P\}$ and $\{V \wedge \neg B\}S2\ \{P\}$. According to their rules, the correctness proposition $\{V\}S\{P\}$ will be true if the following two correctness propositions are true:

[1] $\{V \wedge B\} \Rightarrow P^{z}_{x}$
[2] $\{V \wedge \neg B\} \Rightarrow P^{z}_{y}$

Proof of [1]:

$$
\begin{aligned}
& \{V \wedge B\} \\
= \; & \\
& x>0 \wedge y>0 \wedge z>0 \wedge x<y
\end{aligned}
$$

$\Rightarrow$

$x>0 \wedge y>0 \wedge z>0 \wedge \min(x, y)=x$

$\Rightarrow$

P^{z}_{x} ■

Proof of [2]:

$\{V \wedge \neg B\}$

$=$

$x>0 \wedge y>0 \wedge z>0 \wedge x\geq y$

$\Rightarrow$

$x>0 \wedge y>0 \wedge z>0 \wedge \min(x, y)=y$

$\Rightarrow$

P^{z}_{y} ■

To prove the program's total correctness using Baber's approach, the following three correctness propositions need to be proved:

[1] $V \Rightarrow (x<y)\in\{\text{false, true}\}$
[2] $\{V \wedge B\} \Rightarrow P^{z}_{x}$ and $x\in$Set."z"
[3] $\{V \wedge \neg B\} \Rightarrow P^{z}_{y}$ and $y\in$Set."z"

In order to prove them, sufficient information about the sets associated with x, y and z must be added to V. Note that in Baber's total correctness approach, one must explicitly specify the sets associated with variables, instead of leaving ambiguity to programmers.

- In Dijkstra's approach, one starts the proof from the postcondition and works all the way backward to the precondition.

wp (S, P)

$=$

$D(B) \wedge ((B \wedge wp(\text{"z:=x"}, P)) \vee (\neg B \wedge wp(\text{"z:=y"}, P)))$

$=$

$D(B) \wedge ((x<y \wedge wp(\text{"z:=x"}, P)) \vee (x\geq y \wedge wp(\text{"z:=y"}, P)))$

$=$

$$
\begin{aligned}
&\quad D(B) \wedge ((x<y \wedge x>0 \wedge y>0 \wedge z>0 \wedge \min(x, y)=x) \\
&\qquad\qquad \vee (x\geq y \wedge x>0 \wedge y>0 \wedge z>0 \wedge \min(x, y)=y)) \\
&= \\
&\quad x>0 \wedge y>0 \wedge z>0 \\
&= \\
&\quad V \blacksquare
\end{aligned}
$$

- In relational approaches, the intended function computed by this program will be given.

  For a deterministic program, such as S, the verification strategies in Mills' and Parnas' approaches would be similar. Since Parnas does not require a specific format for verification, we will use Mills' to illustrate the approach. *Concurrent assignments* and *conditional rules,* which are defined in 3.1.1, will be used. The format of proof follows [LMW79, MBG87]. The intended function is given as:

  $f = (x>0 \wedge y>0 \wedge z>0 \rightarrow z{:=}\min(x, y))$.

  By the definitions of assignment statements and conditional composition, we have program functions: $b = (x<y)$, $g = (z{:=}x)$ and $h = (z{:=}y)$.

```
proof
    iftest true prove $(b \rightarrow f) = (b \rightarrow g)$
                $(x<y \rightarrow z{:=}x) = (x<y \rightarrow z{:=}\min(x, y))$
    pass

    iftest false prove $(\neg b \rightarrow f) = (\neg b \rightarrow h)$
                $(x\geq y \rightarrow z{:=}y) = (x\geq y \rightarrow z{:=}\min(x, y))$
    pass
result
```

- **Loop constructs**

The loop construct we discussed here is the "**while** B **do** S **endwhile**" structure commonly used in structured programming.

Floyd	Floyd provided the definition of the for loop [Flo67], which can be transformed for the while loop: $VC = (I \wedge B \Rightarrow P_{\Sigma})$, where I represents loop invariant, Σ represents the loop body and P_{Σ} is the postcondition of Σ.
Hoare	$\dfrac{\{I \wedge B\} S\ \{I\}}{\{I\} \text{ while B do S endwhile } \{I \wedge \neg B\}}$, where I represents loop invariant. [HW73]
Dijkstra	$H_0(P) = D(B) \wedge \neg B \wedge P$, is the set of states in which execution of the while loop terminates in 0 iterations with P true, as the guards are initially false; For k>0, $H_k(P) = D(B) \wedge B \wedge wp\ (S, H_{k-1}(P))$; $wp\ (while, P) = \{\exists k: k \geq 0 \mid H_k(P)\}$, terminates after exactly k iterations of the body of the loop. [Dij76]
	$H_0(P) = (D(B) \wedge (\neg B \Rightarrow P))$; For k>0, $H_k(P) = (D(B) \wedge (B \Rightarrow wlp\ (S, H_{k-1}(P))))$; $wlp\ (while, P) = \{\exists k: k \geq 0 \mid H_k(P)\}$.
Baber	A proof rule for total correctness: If $I \Rightarrow (B \in \{false, true\})$ and $I \Rightarrow (B \Rightarrow 0 < var)$ and {I and B and var = var'} S {I and var ≤ var'- ε} strictly then {I} while B do S endwhile {I and not B} strictly, where var is the loop variant[4] function, var' is the initial value of this function, ε is a positive constant. Each execution of S reduces the value of var by at least the fixed amount ε. [Bab02]

(*continued on next page*)

[4] A loop variant is an expression whose value
a) is decreased (or increased) by at least a fixed amount by each execution of the body of the loop
b) has a lower (or upper) bound. [Bab02]

	Sometimes termination is not required. To handle such situations, a semistrict precondition is defined which does not guarantee that loops terminate but otherwise ensures that each statement in the program segment executes with a defined result: If $I \Rightarrow (B \in \{false, true\})$ and {I and B} S {I} semistrictly then {I} while B do S endwhile {I and not B} semistrictly. [Bab02]
	The proof rule for partial correctness is the same as Hoare's: If {I and B} S {I} then {I} while B do S endwhile {I and not B}. [Bab02]
	The proof rule for the while loop with initialization – total correctness: If {V} init {I} strictly and $I \Rightarrow (B \in \{false, true\})$ and $I \Rightarrow (B \Rightarrow 0 < var)$ and {I and B and var = var'} S {I and var ≤ var'-ε} strictly and I and not B ⇒ P then {V} init; while B do S endwhile {P} strictly. [Bab02]
	The proof rule for the while loop with initialization – partial correctness: If {V} init {I} and {I and B} S {I} and I and not B ⇒ P then {V} init; while B do S endwhile {P}. [Bab02]
Mills	The program function for a while loop: [while *b* do *g* od] $=\{(X, Y) \mid (b(X) \rightarrow Y= f \circ g(X)) \wedge (\sim b(X) \rightarrow Y=X)\}$, where X is the initial state, Y is the final state. [Mil75a, LMW79]

(*continued on next page*)

Parnas	Let L_0 be the LD-relation when B is false, L' be the LD-relation for the program executed when B is true, $L_{while} = L_0 \cup (L' \circ L_{while})$ R_{L0} describes the relation when the starting states contain $\neg$B. C_{L0} contains a state in which B is false. R_{Lwhile} describes the possible starting and stopping states when there are at most *i* executions of loop body followed by an execution of $\neg$B. C_{Lwhile} is the set of states in which there can be at most *i* executions of loop body and then a state in which B is false selected. [Par83]

Table 4.9: While loops

Example 4.2: Verification for a loop program in pre/postcondition and function/relation approaches. In this example, a program segment **Sum** [Bab02] calculates the sum of the elements of an array x(1…n), whenever it terminates.

```
program Sum:    sum:=0; i:=0
                While i<n do
                        i:= i+1;
                        sum:= sum+x(i)
                endwhile
```

- In pre/postcondition approaches, a correctness proposition to be verified is:

{V}**Sum**{P}, where

V: (a) $n \in Z \wedge 0 \leq n$

(b) the index bounds of x are 1 and n

P: $sum = \sum_{k=1}^{n} x(k)$

To give a complete specification, Baber uses an additional correctness proposition, as follows, to specify variables that are not allowed to change:

For every data environment d in the domain of **Sum**,
Sum.d = d except for the values of sum and i.

Its proof is usually done informally through inspection.

The loop invariant of this program segment is specified as the following:

I: $n \in Z \wedge i \in Z \wedge 0 \le i \le n \wedge sum = \sum_{k=1}^{i} x(k)$

- Floyd's strategy is that one deals with loops by cutting loops, and then proves each loop-free execution path:
  1) Cut each loop into decision-decision paths with an intermediate point Φ, which is after the initialization and before reaching the loop condition;
  2) Associate an inductive assertion LOOP(n, i, sum) with Φ. This assertion can also serve as the loop invariant.

  What we need to prove is:

  Path1 The assertion is true when Φ is first reached after starting loop execution (i.e., verification condition LOOP(n, 0, 0) holds),

  Path2 The assertion remains true from Φ to Φ (i.e., verification condition LOOP(n, i, sum) $\wedge$ i<n $\Rightarrow$ LOOP(n,i+1,sum+x(i+1)) holds) and,

  Path3 The postcondition of the loop is true when the loop terminates (i.e., verification condition LOOP(n, i, sum) $\wedge$ i≥n $\Rightarrow$ $sum = \sum_{k=1}^{n} x(k)$ holds).

LOOP(n, i, sum) is given as $n \in Z \wedge i \in Z \wedge 0 \le i \le n \wedge sum = \sum_{k=1}^{i} x(k)$, then

Path1 LOOP(n, 0, 0) is $n \in Z \wedge i \in Z \wedge i=0 \wedge sum=0$.

Path2 Assume LOOP(n, i, sum) and i<n,

i.e., $n \in Z \wedge i \in Z \wedge 0 \le i < n \wedge sum = \sum_{k=1}^{i} x(k)$.

LOOP(n,i+1,sum+x(i+1))

=

$n \in Z \wedge i \in Z \wedge 0 \le i+1 \le n \wedge sum + x(i+1) = \sum_{k=1}^{i+1} x(k)$

=

$n \in Z \wedge i \in Z \wedge -1 \le i \le n-1 \wedge sum + x(i+1) = \sum_{k=1}^{i+1} x(k)$

$\Leftarrow$

$n \in Z \wedge i \in Z \wedge 0 \leq i < n \wedge sum = \sum_{k=1}^{i} x(k)$

=

$LOOP(n, i, sum) \wedge i < n$ ■

Thus, $LOOP(n, i, sum) \wedge i < n \Rightarrow LOOP(n, i+1, sum+x(i+1))$ holds.

Path3 Assume $LOOP(n, i, sum) \wedge i \geq n$, i.e.,

$n \in Z \wedge i \in Z \wedge 0 \leq i \leq n \wedge sum = \sum_{k=1}^{i} x(k) \wedge i \geq n$

=

$n \in Z \wedge i \in Z \wedge i = n \wedge sum = \sum_{k=1}^{i} x(k)$

$\Rightarrow$

$sum = \sum_{k=1}^{n} x(k)$ ■

Thus, $LOOP(n, i, sum) \wedge i \geq n \Rightarrow sum = \sum_{k=1}^{n} x(k)$ holds.

This approach can be illustrated by Figure 4.1.

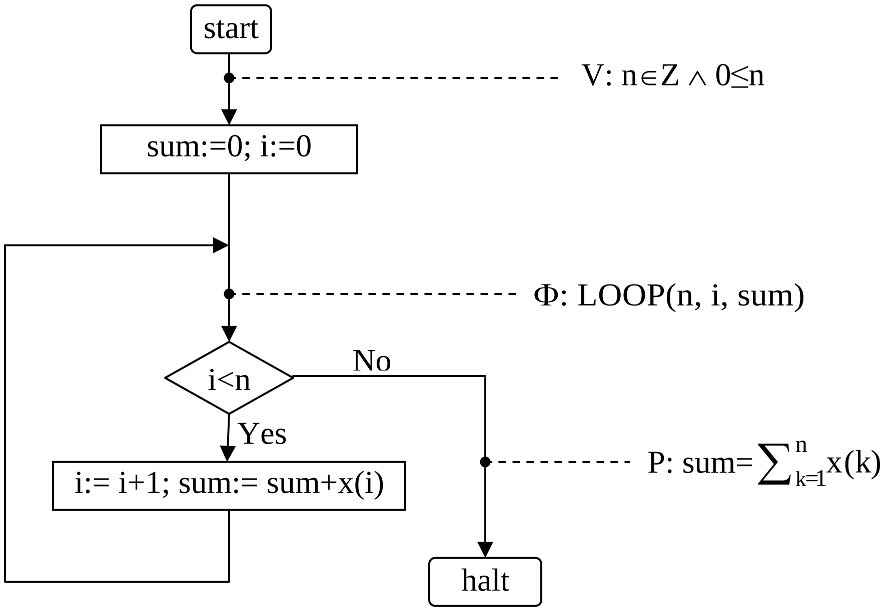


Figure 4.1: Verification using Floyd's approach

- Hoare and Baber use similar rules, which are based on the loop invariant. Baber provides more intuitive and compact proof rules for decomposing the proof of the program. Here we use Baber's partial correctness rule of initialized loops. The propositions to be verified will be as follows:

  [1] $\{V\}$ sum:=0; i:=0 $\{I\}$
  [2] $\{I \wedge i<n\}$ i:=i+1; sum:=sum+x(i) $\{I\}$
  [3] $I \wedge \neg(i<n) \Rightarrow P$

  According to the rule of sequence of assignment statements,
  [1] will be true if [4] holds,

  [4] $V \Rightarrow [I^{i}_{0}]^{sum}_{0}$.

  [2] will be true if [5] holds,

  [5] $I \wedge i<n \Rightarrow [I^{sum}_{sum+x(i)}]^{i}_{i+1}$.

  Then the program can be verified by proving [3], [4] and [5].

  Proof of [3]:

  $$I \wedge \neg(i<n)$$
  $$=$$
  $$n\in Z \wedge i\in Z \wedge 0\le i \le n \wedge sum=\sum_{k=1}^{i} x(k) \wedge i\ge n$$
  $$=$$
  $$n\in Z \wedge i\in Z \wedge i=n \wedge sum=\sum_{k=1}^{i} x(k)$$
  $$\Rightarrow$$
  $$sum=\sum_{k=1}^{n} x(k)$$
  $$=$$
  $$P \blacksquare$$

Proof of [4]:

$[I^{i}_{0}]^{sum}_{0}$

$=$

$[[\, n\in Z \wedge i\in Z \wedge 0\le i \le n \wedge sum=\sum_{k=1}^{i} x(k)]^{i}_{0}]^{sum}_{0}$

$=$

$n\in Z \wedge 0\in Z \wedge 0\le 0\le n \wedge 0=\sum_{k=1}^{0} x(k)$

$=$

$n\in Z \wedge 0\le n$

$=$

V ■

Proof of [5]:

$[I^{sum}_{sum+x(i)}]^{i}_{i+1}$

$=$

$[n\in Z \wedge i\in Z \wedge 0\le i\le n \wedge sum+x(i)=\sum_{k=1}^{i} x(k)]^{i}_{i+1}$

$=$

$n\in Z \wedge i\in Z \wedge 0\le i+1\le n \wedge sum+x(i+1)=\sum_{k=1}^{i+1} x(k)$

$=$

$n\in Z \wedge i\in Z \wedge -1\le i\le n-1 \wedge sum+x(i+1)=\sum_{k=1}^{i+1} x(k)$

$\Leftarrow$

$n\in Z \wedge i\in Z \wedge 0\le i<n \wedge sum=\sum_{k=1}^{i} x(k)$

$=$

$I \wedge i<n$ ■

When we use Baber’s total correctness rule, the propositions to be verified are:

[1] {V} sum:=0; i:=0 {I}strictly

[2] $I \Rightarrow (B\in\{false, true\})$

[3] $I \Rightarrow (B \Rightarrow 0<n\text{-}i)$
[4] $\{I \wedge i<n \wedge (n\text{-}i)=(n\text{-}i)'\}$ i:=i+1; sum:=sum+x(i) $\{I \wedge (n\text{-}i)\leq(n\text{-}i)'\text{-}1\}$strictly
[5] $I \wedge \neg(i<n) \Rightarrow P$

According to the total correctness rule of sequence of assignment statements,
[1] will be true if [6] holds,

[6] $V\Rightarrow [I^{i}_{0}]^{sum}_{0} \wedge 0\in$Set."i" $\wedge$ $0\in$Set."sum".

[4] will be true if [7] holds,

[7] $I \wedge i<n \wedge (n\text{-}i)=(n\text{-}i)'\Rightarrow [[I\wedge (n\text{-}i)\leq(n\text{-}i)'\text{-}1)]^{sum}_{sum+x(i)}]^{i}_{i+1}$
$\wedge\ i+1\in$Set."i" $\wedge$ sum+x(i+1)$\in$Set."sum".

Then the program can be verified by proving [2], [3], [5], [6] and [7]. As we encountered in Example 4.1, these cannot all be proved based on given pre/postconditions. Sufficient information about the sets associated with both *i* and *sum* must be added to both V and I. In fact, this indicates that *i* and *sum* could be global variables, which are contained in the commonly accessible data environment.

- In Dijkstra's approach, one starts from the postcondition as follows.

wp (Sum, P)
=
wp ("sum:=0; i:=0", wp ("while", P))

Here, to compute wp ("while", P), i.e., verify the loop, we cannot use the recursive approach presented in Table 4.9, which will only generate a disjunction as follows:

wp = $H_0(P) \vee H_1(P) \vee \ldots \vee H_k(P)$,
where $H_0(P) = D(B) \wedge \neg B \wedge P$,
$H_1(P) = D(B) \wedge B \wedge wp\ (S, H_0(P))$,
…
$H_k(P) = D(B) \wedge B \wedge wp\ (S, H_{k\text{-}1}(P))$,
and k is the times of iteration of the loop body.

In order to verify the loop, we need the loop invariant I, i.e., wp (“while”, P). Therefore, we need to prove:

[1] $\{wp (\text{“while”}, P) \wedge i<n\}\ S\ \{wp (\text{“while”}, P)\}$
[2] $wp (\text{“while”}, P) \wedge \neg(i<n) \Rightarrow P$

We have proved them in Hoare/Baber’s approach, so we omit the proof process here. Thus,

$$wp (Sum, P)$$
$$=$$
$$wp (\text{“sum:=0; i:=0”}, wp (\text{“while”}, P))$$
$$=$$
$$wp (\text{“sum:=0; i:=0”}, n\in Z \wedge i\in Z \wedge 0\leq i \leq n \wedge sum=\sum_{k=1}^{i} x(k))$$
$$=$$
$$wp (\text{“sum:=0”}, wp (\text{“i:=0”}, n\in Z \wedge i\in Z \wedge 0\leq i \leq n \wedge sum=\sum_{k=1}^{i} x(k)))$$
$$=$$
$$wp (\text{“sum:=0”}, n\in Z \wedge 0\leq n \wedge sum=0)$$
$$=$$
$$n\in Z \wedge 0\leq n$$
$$=$$
$$V \blacksquare$$

Note that, the proofs presented in Floyd’s, Hoare/Baber’s and Dijkstra’s approaches do not explicitly show that the index of *x* are within the bounds 1 and *n*. Hoare [Hoa71a] suggested that one can prove it by using the loop invariant. Alternatively, in Baber’s total correctness approach, the set associated with *i* provides the base for the proof of the boundary values.

- In functional/relational approaches, the intended functions of the program and the program statements will be given as *f*, *f1*(as we mentioned in Example 4.1, we use Mills’ notations here):

Function for the program: $f = (0 \le n \rightarrow i, sum := n, \sum_{k=1}^{n} x(k))$,

Function for the loop: $f1 = (0 \le i \le n \rightarrow i, sum := n, sum + \sum_{k=i+1}^{n} x(k))$.

By the definitions of assignment statements, sequential composition and while loops, we have program functions:

init = (i, sum:= 0, 0),
b = (i<n), and
g = (i, sum:= i+1, sum+x(i+1)).

In Mills' approach, only those variables, whose values are changed, are specified. In Parnas' approach, unchanged variables are explicitly specified by using NC. NC is a predicate symbol which means "Not Changed". It is defined in [IMP93].

proof

(1) **Whiledo** proof:

a) Termination:
Initially 0≤i≤n and i is increased by 1 each iteration, so eventually whiletest i<n fails.

b) **whiletest true**: prove $(b \rightarrow f1) = (b \rightarrow f1 \circ g)$, i.e., $(b \rightarrow (f1 = f1 \circ g))$.

$b \rightarrow f1$

=

$i<n \rightarrow (0 \le i \le n \rightarrow i, sum:=n, sum+\sum_{k=i+1}^{n} x(k))$

=

$i<n \rightarrow i, sum:=n, sum+\sum_{k=i+1}^{n} x(k)$

$b \rightarrow f1 \circ g$

=

$i<n \rightarrow f1 \circ (i, sum:= i+1, sum+x(i+1))$

=

$i<n \rightarrow (0 \le i+1 \le n \rightarrow i, sum:=n, sum+x(i+1)+\sum_{k=i+2}^{n} x(k))$

=

$i<n \rightarrow i, sum:=n, sum+\sum_{k=i+1}^{n} x(k)$

Thus, $(b \rightarrow f1) = (b \rightarrow f1 \circ g)$ holds.

Pass

c) **whiletest false**: prove $(\sim b \rightarrow f1) = (\sim b \rightarrow I)$, where *I* is an identity function.

Case c1: i>n

$\sim b \rightarrow f1$

=

$i>n \rightarrow (f1 = \text{undefined})$

$\sim b \rightarrow I$

=

$i>n \rightarrow (I = \text{undefined})$

Thus, $(\sim b \rightarrow f1) = (\sim b \rightarrow I)$ holds in Case c1.

Case c2: i=n

$\sim b \rightarrow f1$

=

$(i=n \rightarrow (0 \le i \le n \rightarrow i, sum:=n, sum+\sum_{k=i+1}^{n} x(k)))$

=

$(i=n \rightarrow i, sum := n, sum+\sum_{k=n+1}^{n} x(k))$

=

$(i=n \rightarrow i, sum := i, sum)$

=

$\sim b \rightarrow I$

Thus, $(\sim b \rightarrow f1) = (\sim b \rightarrow I)$ holds in Case c2.

pass

Note that in functional/relational approach, one does not need a loop invariant. Instead, a loop function will serve well in the verification.

(2) Sequence proof: prove $f = f1 \circ init$

$f1 \circ init$

=

$f1 \circ (\mathrm{i, sum{:=}0, 0})$

=

$0 \le 0 \le n \rightarrow \mathrm{i, sum{:=}n}, 0 + \sum_{k=1}^{n} x(k)$

=

$(0 \le n \rightarrow \mathrm{i, sum{:=}n}, \sum_{k=1}^{n} x(k))$

=

f

Thus, $f = f1 \circ init$ holds.

pass

result

Discussion

- Floyd's approach, which preceded and influenced Hoare's approach and consequently Baber's and Dijkstra's, addresses the loop iteration by means of an invariant condition. A loop invariant is a condition that holds at the beginning and at the end of the loop. Dijkstra used loop invariants for verification as well although they are not presented in semantic definitions of loops in *wp*. Baber suggests that finding the loop invariant should be the first step to finding a correct loop and it is easy to write the program when you know the invariant.

  Mills gave a recursive definition for the meaning of an iteration statement [LMW79, MBG87]. This idea goes back to McCarthy [McC63]. In Mills' approach, instead of using a loop invariant, a loop function will be needed in verification (i.e. *f1* in the Example 4.2). Then, the function must be shown to be equivalent to the expanded version of the loop.

Formulating loop functions and formulating loop invariants are very similar tasks. In either approach, initialized loops are easier to handle because one can take advantage of the initialization information to simplify the description.

- They all use the same strategy to prove the termination. One first finds a loop variant or a monotonically decreasing quantity (MDQ), and then proves that it decreases along each execution and when it hits the bound the program will stop. Mills presented a more sophisticated version in [Mil75a]. Essentially he showed that in an iteration equation, i.e., f = while b do g endwhile, "the predicate b is fixed entirely by f alone". He called b the *iteration derivative* of f. Baber formulated the strategy as part of the proof rule as shown in Table 4.9. His formulation makes the concept easier to follow during verification.

  Most methods require the loop variant to be an integer, but not in the Baber's method. Baber permits a non-integer variant for proving termination of loops. Allowing the variant to be any real number is advantageous for some loops, e.g. for finding an approximation to the zero of a function.

- All the program constructs we discussed are deterministic. Non-determinism is also an important issue to consider. Both Dijkstra and Parnas introduce languages consisting non-deterministic constructs [Dij75, Par83], which can be used for modeling non-deterministic behaviours. Dijkstra used *wp* to define his guarded command language. Parnas defines his language using LD-relations.

- **Procedure call**

For practical applications, it is important to support procedure calls in verification. Procedure calls are complicated by things like various parameter passing mechanisms, side effects, and the order of evaluation of expressions and parameters.

Floyd and Dijkstra did not consider procedure calls in their approaches.

- **A procedure call without formal parameters**

For a parameterless procedure call, Hoare, Baber, Mills and Parnas use similar manner to handle it: since the procedure call without parameters is semantically equivalent to the body of the procedure, the verification of the call will be equivalent to the verification of the body, which can be conducted through the verification of the statements constituting the body. This assumption is not necessary true in all languages but was valid in most commonly used languages.

Baber provides some rules on it and those rules are basically notational variants of the rules applied for program statements [Bab02]. Hoare provides an inference rule: $\frac{p\ proc\ \ \mathrm{S}, \{\mathrm{V}\}\mathrm{S}\{\mathrm{P}\}}{\{\mathrm{V}\}\ \mathrm{call}\ \ p\,\{\mathrm{P}\}}$ [Hoa71b], here "*p proc* S" represents that *p* has been declared as a parameter-free procedure. *S* is the procedure body.

- **A procedure call with formal parameters (Call by value, Call by name, Other)**

Hoare

The rule of substitution below is used to handle parameterized procedure calls:

$$\frac{\{\mathrm{V}\}\ \mathrm{call}\ p\,(c)\ (n)\,\{\mathrm{P}\}}{\{\mathrm{V}^{f\,c\,n}_{f'a\,m}\}\ \mathrm{call}\ \ p\,(a)\ (m)\,\{\mathrm{P}^{f\,c\,n}_{f'a\,m}\}} \quad \text{[Hoa71b]}$$

Here *c* and *n* represent the lists of formal parameters which change and do not change respectively. *a* and *m* represent the actual parameters. *f* is a list of all symbols which are free in *V* and *P*, must appear in actual parameters, and do not appear in formal parameters. Any of *f* that appears in *V* or *P* must be replaced with completely new symbols *f'*. The substitution of *f* must be performed first.

This rule can handle call-by-value and call-by-reference theoretically. However many assumptions must hold in order to prevent aliasing problems or naming conflicts [Hoa71b]. In [Hoa71b], Hoare presented several rules for procedure calls to apply his axiomatic method to the procedure and parameter passing features of a programming language. The above one is more powerful than the others. He also proposed the rule for functions in [CH72] and [ACH76], in which he corrected inaccuracies of previous rules. Because of the restrictions and complexity, these rules are hard to be applied.

For example, we use *call-by-reference* parameter passing mechanism, which is used in FORTRAN, C++ and Java, in the following program:

```
void swap (int &a, int &b) {
        a:=a+b;
        b:=a-b;
        a:=a-b
}

main( ){
        int x=1, y=2;
        swap(x, x);
}
```

Due to the limitation, Hoare's rule of substitution cannot handle the verification of the call to *swap*. Both the actual parameter lists *a* and *m* contain the same variable *x*. This situation is disallowed when applying his rule because the proof of the body of the subprogram is no longer valid.

Baber

Instead of using complicated rules to handle various procedure calls, Baber models it by equivalent combinations of the program statements.

One can use declaration and release for proper variables to explicitly implement the semantics of a particular parameter passing mechanism. Then procedure calls with parameters may be replaced by the equivalent parameter-free procedure calls. By using rules of declaration and release, verification of parameter passing can be handled explicitly and be prevented from getting into too much complexity.

For *call-by-value*, new, local variables are declared whose initial values are the values of actual parameters and are released at the end of the procedure. For *call-by-value-result*, it is much like dealing with the call-by-value, but in addition the values of formal value-result parameters have to be assigned back to actual value-result parameters. For *call-by-name,* since the names of formal parameters are changed throughout the procedure to the names of actual parameters and the resulting procedure will be executed, it can be solved through changing formal parameters' names to actual parameters' names in the original procedure [Bab87, Bab02]. *Call-by-name* mechanism plays a major role in ALGOL-60. However, due to its considerable execution overhead, it is not widely used [PZ01]. Baber's approach does not consider *call-by-reference*.

Implementations of parameter passing mechanisms have many variants in actual systems. One must fully understand the mechanisms in the language being used in order to apply the method accurately.

Suppose the above example of *swap* implemented using *call-by-value-result* and *call-by-name* mechanism respectively. By Baber's approach, we have

1) call-by-value-result

The call {x=1}*swap*(*x*, *x*){x=1} is equivalent to the call without parameters {x=1}*swapx*{x=1} where the subprogram *swapx* is

```
                                         {x=1}
declare (a, Z, x); declare (b, Z, x)
                                         {b=1 ∧ a=1}
a:=a+b
                                         {b=1 ∧  a-b=1}
b:=a-b
                                         {a-b=1 ∧ b=1}
a:=a-b
                                         {a=1 ∧ b=1}
x, x:=a, b
                                         {x=1}
release a, b
                                         {x=1}
```

By applying proof rules, which are shown on the right side, we know that *swapx* satisfies its pre/postcondition. Therefore the call *swap*(*x*, *x*) using *call-by-value-result* will execute correctly.

2) call-by-name

The call {x=1}*swap*(*x*, *x*){x=1} is equivalent to the call without parameters {x=1}*swapx*{x=1} where the subprogram *swapx* is

```
x:=x+x
x:=x-x
        {x-x=1} contradiction
x:=x-x
        {x=1}
```

It is obvious that *swapx* does not satisfy the pre/postcondition required. The procedure call will fail to execute correctly.

We know that different parameter passing mechanisms may affect the result of execution completely. This example is a typical aliasing problem. It happens easily with passing parameters, such as two parameters aliased or a parameter and a global variable aliased. Baber’s guidelines provide general ideas on passing parameters, but do not deal with aliasing problems explicitly.

Mills, Parnas

Neither Mills nor Parnas consider the invocation with parameters. Mills and Parnas assumed that one knows the program function of each invocation of a procedure.

Discussion

Even a subtle difference in parameter passing mechanism may affect the result of execution. Rules will not give much help for the verification or documentation of procedure calls. Many different inference rules for procedures calls have been proposed through the years correcting limitations and inaccuracies of previous rules [Apt81]. For these reasons, most people avoid the issue of parameter passing when talking about proof and specification mechanisms.

- **Recursive invocation of programs**

Floyd, Dijkstra ignored this issue.

Hoare

For the recursion in procedure call, Hoare provides a general recursive invocation rule:

$$\frac{p(x)(v)\ \mathrm{proc}\ \mathrm{S}, \{\mathrm{V}\}\ \mathrm{call}\ \ p\ (x)(v)\ \{\mathrm{P}\} \vdash \{\mathrm{V}\}\mathrm{S}\{\mathrm{P}\}}{\{\mathrm{V}\}\ \mathrm{call}\ p(x)(v)\ \{\mathrm{P}\}} \quad [\mathrm{Hoa71b}]$$

It means that “if {V}S{P}can be deduced from *p* (a procedure whose body is *S*) and {V}call *p*(*x*):(*v*){P} is assumed, then {V}call *p*(*x*):(*v*){P} holds”. This rule actually presents a solution using induction. One first assumes that if the pre/postcondition of the recursive call holds, the pre/postcondition of the body can be verified. The base case of the induction is that the part of the body that does not contain the recursive call can be verified. Then the assumption and the base case together can prove the call of recursive procedure.

Baber

One uses the rules for the declare and release statement to handle the stacking and popping of the recursion.

Mills

Instead of using substitutions in ordinary procedure calls, one uses a technique similar to loops. Mills [MBG87] provided following verification rule for recursive invocation:

A recursive procedure:

```
proc P
begin
    if B
    then S1
    else S2; call P; S3
endproc
```

where B is a boolean expression and S1, S2 and S3 are statements that do not call P.
Then *f* = [P] if and only if:

1) domain(*f*) = domain([P])
2) *f* is a solution to the recurrence equation (a recurrence equation is an equation in which an unknown function occurs on both sides [MBG87]).

Parnas

Parnas' approach focuses more on documentation. A program, which is deterministic, computes the same function no matter it computes recursively or iteratively. Thus, when one is documenting a recursive computation, all s/he states is a black box function.

Discussion

We have presented various rules proposed in each approach. It should be noted that these rules differ from language to language. All of the researchers did their work in terms of an idealized language that they believed was typical or indicative of how one does it with a real language. However, real languages differ in such things as the scope rules and parameter passing so the rules must be used carefully.

4.2 How does the approach support verification of a program?

While most of the approaches we compared support verification, they use different strategies and procedures as shown in Examples 4.1 and 4.2.

In pre/postcondition approaches,

1) The verification goal is to prove the truth of {V}S{P} which means that if V is true before S is executed, P must be true after the execution is completed. V and P are given as the specification of the program.
2) Verifying the program by: a) applying proof rules iteratively to decompose the correctness proposition to be proven until primitive statements are reached; b) proving correctness propositions. Dijkstra's approach uses a backward strategy, (i.e., starting from the postcondition P).
3) For each loop in the program, an *invariant condition* is given. By using a loop invariant, one can prove that a loop will always generate desired results.
4) If termination is required, one uses a loop variant or a monotonically decreasing quantity to prove it.

In relational approach,

1) The verification goal is to show that the intended relation (i.e., specification) is equivalent to the relation that the program computes.
2) The intended relations for program segments and major parts of the code, such as loops, should be given. One generates program relations that describe the program, then proves the equivalence of intended relations and program relations for each segment and the whole program.
3) If termination is required, one uses a monotonically decreasing quantity to prove it.

Floyd actually used different strategy than other pre/postcondition approaches as we illustrated in Example 4.1 and 4.2. He used the assertions and the program to formulate the theorems, then proved those theorems in the mathematical domain. This strategy is similar to Mills' approach. To verify a program, Mills derived theorems based on intended functions and the program, then prove the theorems without ever using the program. Other pre/postcondition approaches usually work between program domain and mathematics domain. They go through the program proving as they go.

Parnas also proposed the Display Method for documenting the program [PMI94, BP95, Pet96] to assist verification and inspection. In display method, the programs and their specifications are organized into a set of displays, each containing:

1) the specification for that program.
2) the program text.
3) the specifications for all of the other programs invoked within that program.

Through checking individual displays, one uses the "divide and conquer" technique to examine each program in isolation. Display method can be used by other approaches as well. An example of displays in Baber's approach is presented in Appendix A.

4.3 Side effects handled?

One of the semantically difficult features which appears early and often in software verification is handling expressions which have side effects. A side effect occurs when an expression has an effect – modifying the state of a system – from only producing a value.

In pre/postcondition approaches, one associates assertions with the states before and after the execution of a program. Without caution and proper treatment, it is inevitable that the side effects associated with a statement will make the formulations of its initial or final states invalid. Historically there have been two general approaches to handling programming languages whose statements may have side effects [Hom95, Bla98]. One [Boe85] is by having separate inference rules – "effect" rules for the effect of expressions on the state; "value" rules for the result value of expressions or statements. The other approach [CG76, Kow77] is to analyze the subexpressions which cause side effects separately from the original expression and uses unique variables which carry the result of side effects. Although the former was found to be easier to apply [Bla98], both approaches will increase the complexity of the formalization. In the approaches we compare, it is assumed that expressions are evaluated without side effects. They assume that execution of a statement may change only the variables indicated and the evaluation of an expression may change no variables. Floyd [Flo67] gave an example of a simple treatment for the side effects of the extended assignment statement which allows embedded assignments as subexpressions.

Relational approaches have no problem with side effects. With knowing the function/relation of each individual component in an expression and the sequence of operations (i.e., how the expression is evaluated), the effects/results can be clearly expressed. Side effects are actually included in the relation. No special treatment is needed.

4.4 How does the notation of an approach facilitate its use in documentation?

Notations are critical for documentation throughout the whole process of software development. Documents should be written in a precise and accessible way. A carefully chosen notation can reduce the effort tremendously in the process of preparing and reviewing these documents [JPZ95]. We compare the proposed notations of each approach by following criteria:

1) Accessibility: i.e. ease of learning. The documentation should avoid using many special notations or formats. The familiarity of the material can stimulate the learner and reduce much effort in learning.
2) Readability: i.e. ease of comprehension, which is influenced by the size and the lucidity of formulae. Good notations will increase the clearness and compactness of formulae. They appeal to the mind and are easy to be understood.
3) Checkability: A mechanism exists that can assist designers/programmers to assure the correctness of documents.

Floyd, Hoare, Dijkstra

Their approaches mostly use mathematical notations, first-order logic operators and imperative programming expressions with a few conventions. While they did not use many special symbols or notations, they do not provide any means to increase the accessibility and readability for the documentation.

Baber

Baber’s notations are mainly from first-order logic and Hoare’s conventions. These are not restricted to the traditional format in order to make the formulae more compact. Baber also substitutes words for mathematical symbols (e.g. “and”, “or”, etc.) to make the formulae more readable.

Baber uses a decomposition process diagram. The decomposition process of verifying a program is usually complex. Verifiers have to pay extra caution to avoid missing any steps. A decomposition process diagram makes it easier to check the whole process and clearly shows the correctness propositions that need to be proved. In the following diagram, the correctness propositions, located in the last node of each branch, need to be proved.

```
Sum:  sum=0; declare (i, Z, 0)
      While i<n do
            i:= i+1;
            sum:= sum+x(i)
      endwhile
      release i
```

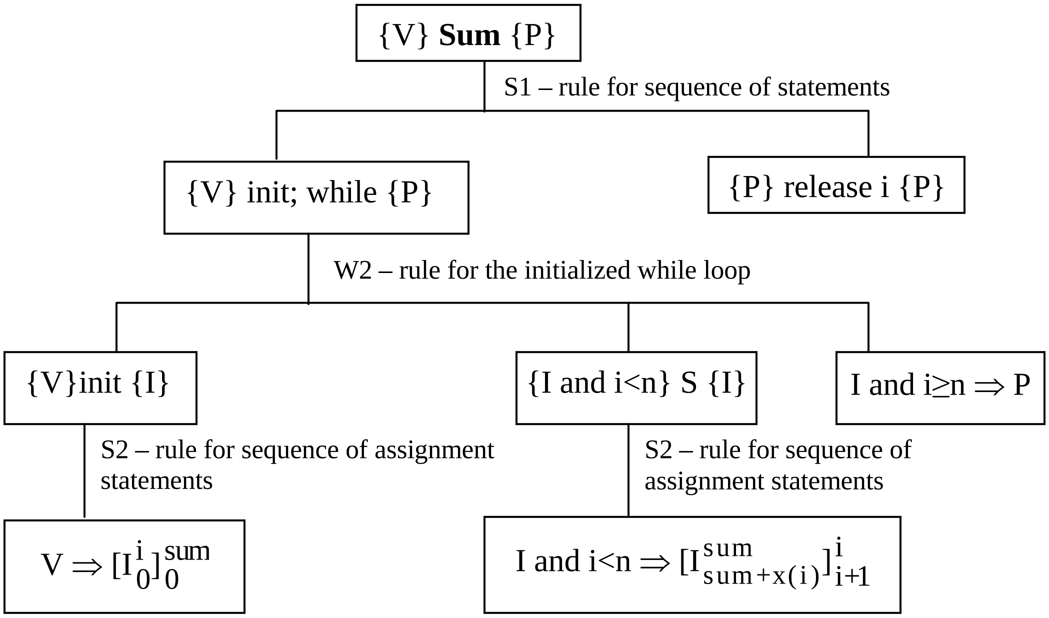

{V} **Sum** {P}
S1 – rule for sequence of statements
{V} init; while {P}
{P} release i {P}
W2 – rule for the initialized while loop
{V}init {I}
{I and i<n} S {I}
I and i≥n ⇒ P
S2 – rule for sequence of assignment statements
S2 – rule for sequence of assignment statements
$V \Rightarrow [I_0^i]_0^{sum}$
I and i<n $\Rightarrow [I_{sum+x(i)}^{sum}]_{i+1}^{i}$

Figure 4.2 An example of decomposition process diagram in Baber's approach

Mills

Mills' functional expressions use conditional rules and concurrent assignments, which are simpler to write and easier to read than the conventional function expressions. Mills also used trace tables to systematically show the symbolic execution in the process of verification [LMW79, MBG87].

Parnas

Parnas uses characteristic predicate expressions in tabular notations. He proposed tabular expressions for relational documentation. Tabular expressions have been shown to be an effective tool for presenting formal computer system documentation in a concise and readable manner.

Documents written in conventional mathematical expressions are often lengthy and deeply nested. Great caution has to be taken during the process of documentation and review. The definition of functions *f*(*x*, *y*) [JPZ95] is written in two formats below.

$$(\forall x, (\forall y, (((x\geq 0\wedge y=10)\rightarrow f(x, y)=0) \wedge ((x<0\wedge y=10)\rightarrow f(x, y)=x) \wedge ((x\geq 0\wedge y>10)\rightarrow f(x, y)=y^2) \wedge ((x\geq 0\wedge y<10)\rightarrow f(x, y)=-y^2) \wedge ((x<0\wedge y>10)\rightarrow f(x, y)=x+y) \wedge ((x<0\wedge y<10)\rightarrow f(x, y)=x-y))))$$

f(x, y) =	y = 10	y > 10	y < 10
x≥0	0	y^2	$-y^2$
x<0	x	x+y	x-y

Figure 4.3: An example of tabular notation

The one using tabular notations is more readable. The expression can be understood intuitively because that intuition is consistent with its formal definition. One can use tables to “parse” formulae instead of parsing in his/her mind. Formal semantics of tabular notation is defined in [HKP78, Par92, Par94b, Par95, JK01].

Even in this simple function, it is easy to overlook cases. Comparing to the expressions written in traditional format, tabular notations are much more suitable for describing the functions, relations and conditions that frequently occur in program specifications and descriptions. Tables make it easy to detect missing cases. Another feature of tabular expressions is that they fully utilize the “divide and conquer” strategy. Inspector can review the problem one column or one row at a time without being distracted by other cases.

In addition, tabular notations are suitable for both application-dependent and application-independent checking. For the application-dependent checking, one checks the content of cells in the main grid. For the application-independent checking, completeness is assured if the union of all the header expression is true and consistency is assured if the intersection of any pair of expression in header is false. More examples of tabular notations are presented in Appendix A and B.

4.5 How does the approach handle program derivation/ design?

Derivation means a mechanical process of producing the desired program. To derive a program means to have a formal specification first, and then apply transformation rules in order to obtain an executable program. The program you obtain is then correct by construction. Mathematical derivation has its advantages in that one just needs to do the calculation in a mechanical way by following some rules, which will be more efficient than guessing and then verifying.

However, derivation also has its limitations. Because of the nature of software, human intuition, creativity or inventiveness are often useful or needed during the process of design. Besides, human decision is also involved to make optimal choice among the alternatives which all satisfy the given specification. A program produced by derivation may be less efficient in actual use than the one done more intuitively.

Floyd did not consider this issue. Hoare gave some consideration on the design of large systems [Hoa87], but did not provide approaches for program design.

Dijkstra

Dijkstra proposed the formal derivation of programs from specifications [Dij75]. Programs are constructed by simultaneously deriving and verifying them from the postcondition. He also proposed a design approach – stepwise refinement [Dij72], which means that a problem is broken into a series of smaller steps, these steps are further refined until the problem can be solved by following all the small steps, from which an algorithm is constructed.

Baber

Baber provides design procedures and a series of guidelines that are intended to be applied informally and intuitively.

When the need for program derivation arises, Baber takes a pragmatic strategy with formal and informal approaches combined. Programmers can follow rules for the derivation (some of the rules given in 4.1), instead of programming in an ad hoc manner [Bab02].

For example, to design a loop, one starts by determining a suitable loop invariant I. This step is conducted informally by producing a generalization of V and P. Then one can derive three parts of the loop – the initialization, the loop condition and the loop body – more formally based on I. According to “$\{V\}$ init $\{I\}$”, the initialization can be derived. Then the loop condition can be found according to “$I \wedge \neg B \Rightarrow P$”. For the loop body, based on “$\{I \wedge B\}$ S $\{I\}$”, one first considers the termination of the loop, then considers to reestablish the truth of I, which is the postcondition of the body S.

Mills

Mills presented the stepwise refinement approach [Mil75b]. One begins with the specification, which given as an intended function. The intended function is repeatedly divided into low-level intended functions (sub-specifications). This decomposition process continues until intended functions are expanded into suitable program structures. Then, the equivalence between the designed program functions and the given functions needs to be verified.

For example, *f* is a given function. We assume an equivalent function will be produced by a *while* structure with function or predicate components *b*, *u*, *v*, hence we need to prove *f* = [while *b* do *u*; *v* endwhile].

Parnas

Parnas did not discuss either formal derivation or stepwise refinement. He proposes general design methods for decomposing programs and module design [Par72], etc. In his view, "design is recognized as a very creative task in engineering fields, in which mathematics and science provide essential inputs, but the primary role of the mathematics comes in the documentation and validation of the design" [Par93b].

4.6 Programming languages limitation

A generic classification of programming language is as follows [PZ01]:

- Imperative programming languages: Its basic concept is machine states, the set of all values for all memory locations in the computer. It is strongly tied to the concept of variables and memory locations.
- Functional programming languages: Its view of the computation is to look at the function that the program represents rather than just the state changes as the program executes. In particular, functional programs do not use the concept of variables in the traditional sense, i.e. a memory location whose contents might be changed from time to time as a program executes.
- Logic programming: Logic programming is based on first-order logic. Programs are collections of facts stated in logic, and programs are executed by an engine that performs proofs based on the stated facts.
- Object-oriented programming languages: Complex data objects are built, then a limited set of functions are designed to operate on those data. They can be seen

as imperative languages with additional features. While languages such as C++ and Java are commonly referred to as object-oriented programming languages, they are also imperative languages.

All the methods we compare were developed for imperative languages. Because of the connection between imperative languages and object-oriented languages, all methods can be applied equally easily to those object-oriented languages.

None of the methods apply to functional languages. All the methods assume that one is dealing with state changes and dealing with variables, while functional programs do not use the concept of variables in the traditional sense. None of the methods apply to logic languages either.

4.7 Does the approach provide explicit descriptions on how to use the transformation/proof rules?

Knowing the rules does not teach how to use the rules. Are the rules presented in a way that they can be followed by practitioners? It is important to show the programmer how to use a mathematical concept, not just to teach them the definitions and theorems [Par93b]. In order to do so, explicit explanations and illustrative examples are needed.

Floyd's, Hoare's and Dijkstra's work were in the early stage of the development of pre/postcondition methods. While they elaborated on how to apply the approaches, increasing the accessibility of the approach had not gain much attention at the time.

Baber's proof rules are similar but not limited to Hoare's. Baber extends each basic rule to provide a specific rule for a particular situation. For instance, Baber defines three rules for assignment statements – A1, A2 and S2. Rule A1 is used to derive a precondition for a given postcondition and a given assignment statement. A2 and S2 act as auxiliary rules with A2 used for verifying a single assignment and

S2 used for verifying a sequence of assignments. These auxiliary rules increase the applicability of the approach.

Examples are one effective means to well describe the usage of rules. In [Bab87], Baber presents examples, which are extracted from productive commercial systems, to illustrate how his approach can be applied to real software of reasonable size. Mills introduced his approach along with elucidating a programming language – PASCAL [MBG87]. The verification rules and procedures were explained and illustrated by applying them to various mechanism of PASCAL. Parnas provides many documentation paradigms using relational approach and tabular expressions. Guidelines for documentation and inspection are also provided.

4.8 Conclusion

While all approaches provide certain rules for the most commonly used program constructs, Parnas' and Dijkstra's approaches consider non-deterministic constructs as well and further provide definitions.

Some complicated programming issues are not fully dealt with in all approaches in terms of verification and specification. These issues include pointers, aliasing problems, parameter passing and scope rules. Baber's approach provides some consideration on scope issue, but not completely treated and defined.

As for side effects, pre/postcondition approaches have to ignore this issue otherwise the complexity of the formulae will be increased considerably. Relational approaches do not have this problem. Side effects have been included during their formalization.

Parnas, Baber and Mills give more consideration to the representation of documentation. Parnas' tabular expression is a powerful means for constructing and presenting program documents.

Dijkstra and Mills provided the stepwise design techniques in their own methods. Dijkstra also proposed the program derivation. Baber’s approach deals with program derivation and also provides many guidelines for program design.

Floyd, Hoare and Dijkstra concerned themselves mostly with the theoretical soundness of the approaches. Accessibility did not gain much attention at their time. Baber, Mills and Parnas give great amount of consideration to advancing the accessibility of the approaches. Baber uses examples, informal descriptions, and diagrams to illustrate the method in a comprehensible manner. Mills gave detailed explanations and various examples of verification. Parnas focuses on the issues of program inspection and documenting programs in an efficient form by using tables.

Chapter 5

Conclusion and Future Work

This chapter summarizes the advantages and limitations of each approach, draws conclusions, provides suggestions and proposes future work.

5.1 Conclusions and suggestions

We have elaborated and illustrated six approaches through comparing their mathematical models, specification and verification techniques as well as applicability and practicability. It was not our goal to find a perfect development method that can be applied equally well in various situations. We intend to find out their relative strengths or weaknesses in terms of practicability. In the long run, one would like to find methods that have the strengths of all and the weaknesses of none but that is beyond the scope of this thesis. The main conclusions of this work are as follows. Some suggestions regarding the application of the approaches are also given.

- Among these approaches, Dijkstra's, Parnas' and Baber's methods have more powerful expressive ability than others' in terms of specifying termination of programs. As discussed in 3.1, there are generally three termination statuses that a program may encounter. Only Dijkstra's, Parnas' and Baber's approaches can specify all three statuses.

- Mills and Parnas distinguish specifications and descriptions through constructing intended functions/relations and program functions/relations. Floyd, Hoare, Dijkstra and Baber's approach did not make such distinction. However, by following the guidance provided in Parnas' approach, one can also write program descriptions using pre/postconditions through inspecting programs and applying transformation rules.

- Dijkstra's and Parnas' approaches explicitly consider non-determinism and can deal with both deterministic and non-deterministic programs, whereas other approaches chose not to stress this aspect. Baber also includes non-deterministic specifications into his model. In Section 3.3.1, we pointed out that non-determinism are not intended in other approaches.

- By defining a specific state representation and program variables, Baber's approach is able to deal with some practical problems, such as variable binding, variable scope, procedure calls and recursive invocations. Mills and Parnas do not introduce a state representation into their basic models and have no underlying assumptions on it, hence users can choose whatever representations, e.g., Baber's representation, that is suitable for the problem in hand. Floyd, Hoare and Dijkstra made simplifying assumptions on this issue, which are, however, invalid in real applications.

- Through the discussion about various treatments of some common program statements and constructs in Section 4.1, we revealed a few differences and similarities between the two groups of approaches:

  1) The basic semantics of programs are defined differently in the two groups. As we have seen in Section 4.1.1, pre/postcondition approaches define basic semantics of programs as an inference system. Basic statement types/constructs are defined either as axioms, inference rules or as basic definitions, which are necessary for an inference system. Relational approaches are formulated in terms of mapping. They provide semantic definitions for basic statement types as mapping from a domain of states to a ranges of states. Programming constructs such as "if then else" are defined as

mappings from a domain of mappings (the state-to-state mappings that describe the statements) to other such mappings.

2) One of the important concepts in pre/postcondition approaches is the notion of loop invariants which should be maintained throughout the execution of loops and is essential to verifying loops. However, in some relational approaches, one does not need the loop invariant. Instead, one states the intended function/relation for the loop and the function/relation that describes the components (body, termination condition of the loop) and then checks an equation in mappings to see if it holds.

3) All these approaches do not consider explicitly some detailed issues that are often encountered in programming, such as pointers, dynamic memory allocation, scope rules, etc. They are built on an idealized language in order to simplify formalization. However, to apply these techniques to real applications, these issues have to be tackled for practitioners following the methods. Some approaches already provide the basis to build on. For issues of pointers and dynamic memory allocation, pointers can be handled as indices to arrays and dynamic memory allocation can be handled by the declaration and release of array elements. Both Baber's and Mills' approaches already provide rules of dealing with arrays. Baber's approach also provides a basis for handling scope rules and parameter passing. More systematic techniques could be developed from it.

- Through the discussion on side effects, we know that side effects will not affect relational approaches. Pre/postcondition approaches need special treatment to correctly handle side effects, which may increase complexity greatly, so they assume no side effects occurs.

- Mills, Parnas and Baber put great consideration on helping people to use the approaches. Baber provides detailed guidelines for both design and verification. Parnas introduces an effective documentation technique – tabular expressions – and proposes the Display method. In Appendix B, we present the specifications using both Parnas' and Baber's approaches for the same programs. The good thing is that these techniques are not dependent to a specific method. They can be applied in other approaches as well. Baber's guidelines and diagrams can be used in other pre/postcondition approaches. Tabular expressions could be applied in all the other approaches to represent predicates.

5.2 Future work

In this comparative study, we illustrate the ideas with simple examples. More substantial examples, which may come from the requirement of a real world project, need to be developed to further acquire quantitative results in terms of applicability and efficiency of these approaches.

Another area of future work is to integrate Parnas' and Baber's approaches to use practical advantages of each. They both have good abilities of constructing specifications, which are illustrated in Appendix B. They have the same concern on increasing practicability. In addition, they are complementary to each other in some areas. In Appendix A, we have attempted to use both techniques to solve the problem. A more systematic and compact approach could be discovered, especially with the notion of a postcondition characterizing the relation between the initial and final states. Moreover, empirical examples can be developed at the same time to illustrate the integration.

We have discussed that Baber's and Parnas' methods can specify all three terminating behaviours. But we did not further formalize or prove the relationship between the two approaches, which could be one area of the future work. A connection between the two approaches could be established by formulating and proving a theorem about the relation between a pre/postcondition pair and a LD-relation.

Bibliography

[Abr96] J. R. Abrial, *The B Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.

[ACH76] E. A. Ashcroft, M. Clint and C. A. R. Hoare, “Remarks on program proving: Jumps and functions”, *Acta Informatica*, Vol. 6, 1976, pp. 317-318.

[Air91] M. Mac an Airchinnigh. “Tutorial Lecture Notes on the Irish School of the VDM”, Edited by S. Prehn and W. J. Toetenel, *VDM'91, Formal Software Development Methods: Tutorials (Lecture Notes in Computer Science* 552*)*, Springer-Verlag, Berlin, 1991, pp. 141-237.

[Apt81] K. R. Apt, “Ten years of Hoare's logic: A survey--part I”, *ACM Transactions on Programming Languages and Systems,* Vol. 3, No. 4, październik, 1981, pp. 431-483.

[Ate02] Atelier-B, http://www.atelierb.societe.com, 2002.

[Bab87] R. L. Baber, *The Spine of Software: Designing Provably Correct Software - Theory and Practice*, John Wiley & Sons, Chichester, 1987. http://baber.servehttp.com/Books/Spine.pdf (December, 2004).

[Bab91] R. L. Baber, *Error Free Software: Know-How and Know-Why of Program Correctness*, John Wiley & Sons, Chichester, 1991. German original: Fehlerfreie Programmierung für den Software-Zauberlehrling, R. Oldenbourg Verlag, München, 1990. http://baber.servehttp.com/Books/ErrorFreeSW.pdf (December, 2004).

[Bab95] R. L. Baber, *Praktische Anwendbarkeit mathematisch rigoroser Methoden zum Sicherstellen der Programmkorrektheit*, Walter de Gruyter, Berlin, 1995. http://baber.servehttp.com/Books/PrakAnw.pdf (December, 2004).

[Bab97] R. L. Baber, “Comparison of electrical ‘engineering’ of Heaviside’s times and software ‘engineering’ of our times”, *IEEE Annals of the History of Computing*, Vol. 19, No. 4, 1997, pp. 5-17.

[Bab02] R. L. Baber, *Mathematically Rigorous Software Design*, Electronic textbook, July 2002. http://home.RLBaber.de/Professional/McMaster/Courses/ 46L03/MRSDLect.pdf (December, 2004).

[Bco02] B-Core, URL http://www.b-core.com/aboutbcore.html (December, 2004).

[Bjo82] D. Bjørner, “Stepwise Transformation of Software Architectures”. *Formal Specification and Software Development*, Chapter 11, edited by D. Bjørner and C. B. Jones, Prentice-Hall, 1982, pp. 353-378.

[Bla98] P. E. Black, *Axiomatic Semantics Verification of a Secure Web Server*, Ph.D. dissertation, Brigham Young University, Utah, USA, February 1998.

[Bli87] A. Blikle, “Denotational Engineering or From Denotations to Syntax”, *VDM’87, VDM—A Formal Method at Work*, *Lecture Notes in Computer Science*, Vol. 252, edited by D. Bjørner, et al., Berlin, Springer-Verlag, 1987, pp. 151-209.

[Bli88] A. Blikle, “Three-valued Predicates for Software Specification”, *VDM’88, VDM—The Way Ahead*, *Lecture Notes in Computer Science*, Vol. 328, edited by R. Bloomfield, et al., Springer-Verlag, 1988, pp. 243-66.

[Bli90] A. Blikle, “On Conservative Extensions of Syntax in the Process of System Development”, *VDM’90, VDM and Z—Formal Methods in Software Development*, *Lecture Notes in Computer Science*, Vol. 428, edited by D. Bjørner, et al., Berlin, Springer-Verlag, 1990, pp. 504-525.

[Boe85] H. Boehm, “Side effects and aliasing can have simple axiomatic descriptions”, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, October 1985, pp. 637-655.

[BP95] B. Bauer, D.L. Parnas, "Applying Mathematical Software Documentation - An Experience Report", *Proceedings of the Tenth Annual Conference on Computer Assurance,* National Institute of Standards and Technology, Gaithersburg, Maryland, June 1995, pp. 273-285.

[CG76] R. J. Cunningham and M. E. J. Gilford, “A note on the semantic definition of side effects”, *Information Processing Letters*, Vol. 4, No. 5, February 1976, pp. 118-120.

[CH72] M. Clint and C. A. R. Hoare, “Program proving: jumps and functions”, *Acta Informatica*, 1972, pp. 214-224.

[deB50] N. G. de Bruijn, “On some linear functional equations”, *Publicationes Mathematicae*, Debrecen 1, 1950, pp. 129-134.

[Dij68] E. W. Dijkstra, “Structure of THE-Multiprogramming System”, *Communications of ACM*, Vol. 11, No. 5, 1968, pp. 341-346.

[Dij72] E. W. Dijkstra, “Notes on structured programming”, in *Structured Programming*, edited by O. Dahl, et al., Academic Press, New York, 1972, pp. 1-82.

[Dij75] E. W. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs”, *Communications of the ACM*, Vol. 18, No. 8, August 1975, pp. 453-457.

[Dij76] E. W. Dijkstra, *A Discipline of Programming*, Englewood Cliffs, NJ: Prentice-Hall, 1976.

[Flo67] R. W. Floyd, “Assigning meanings to programs, Mathematical aspects of computer science”, *American Mathematical Society*, edited by J.T. Schwartz, 1967, pp. 19-32. Also in: *Proceedings of a Symposia in Applied Mathematics*, Vol. 19, 1968.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, N.Y., 1981.

[Gur84] Y. Gurevich, “Reconsidering Turing’s Thesis: Toward More Realistic Semantics of Programs”, *Technical Report* CRL-TR-38-84, EECS Department, University of Michigan, 1984.

[Gur93] Y. Gurevich. “Evolving algebras: An attempt to discover semantics”, *Current Trends in Theoretical Computer Science*, edited by G. Rozenberg and A. Salomaa, World Scientific, 1993, pp. 266-292.

[Gur00] Y. Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms", *ACM Transactions on Computational Logic*, Vol. 1, No. 1, 2000, pp.77-111.

[GY76] S. L. Gerhard and L. Yelowitz, "Observations of fallibility in applications of modern programming methodologies", *IEEE Trans. Software Eng.*, Vol. SE-2, No. 3, September 1976.

[GY91] D. I. Good, W. D. Young, "Mathematical Methods for Digital Systems Development", *Proceedings of 4th International Symposium of VDM Europe*, Noordwijkerhout, The Netherlands, October 1991.

[HHH87] C. A. R. Hoare, I. J. Hayes, J. He, et al. "Laws of Programming", *Communications of the ACM*, Vol. 30, No. 8, August 1987, pp. 672-686.

[HJN93] I. J. Hayes, C. B. Jones, J. E. Nicholls, "Understanding the differences between VDM and Z", *Technical Report* UMCX-93-8-1, University of Manchester, August 1993.

[HKP78] K. L. Heninger, J. Kallander, D. L. Parnas, J. E. Shore, "Software Requirements for the A-7E Aircraft", *NRL Memorandum Report 3876,* U.S. Naval Research Lab, 1978.

[Hoa69] C. A. R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, Vol. 12, No. 10, October 1969, pp. 576-580,583.

[Hoa71a] C. A. R. Hoare, "Proof of a program: FIND", *Communications of the ACM*, Vol. 14, No. 1, January 1971, pp. 39-45.

[Hoa71b] C. A. R. Hoare, "Procedures and parameters: An axiomatic approach", *Symposium on Semantics of Algorithmic Languages, Lecture notes in mathematics* Vol. 188, 1971, pp. 102-116.

[Hoa74] C. A. R. Hoare, "Hints on programming language design", *State of the Art Report 20: Computer Systems Reliability*, edited by C. J. Bunyan, Pergamon/ Infotech, 1974, pp. 505-534.

[Hoa87] C. A. R. Hoare, "An overview of some formal methods for program design". *IEEE Computer Journal,* Vol. 20, No. 9, September 1987, pp. 85-91.

[Hom95] P. V. Homeier, *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures*, PhD thesis, Univeristy of California, LA, 1995.

[HW73] C. A. R. Hoare, N. Wirth. “An axiomatic definition of the programming language PASCAL”, *Acta Informatica* 2, 1973, pp. 335-355.

[IMP93] M. Iglewski, J. Madey, D. L. Parnas, P. C. Kelly, “Documentation paradigms (A progress report)”, *CRL Report* 270, Telecommunications Research Institute of Ontario (TRIO), McMaster University, July 1993.

[Jac97] J. Jacky, *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997.

[Jin00] M. Jing, “A Table Checking Tool”, *SERG Report*, No. 384, McMaster University, March 2000.

[JK01] R. Janicki, R. Khedri, “On a Formal Semantics of Tabular Expressions”, *Science of Computer Programming*, Vol. 39, 2001, pp. 189-213.

[JM92] C. B. Jones, A. M. McCauley, “Formal methods - selected historical references”, *Technical Report* UMCS-92-12-2, University of Manchester, 1992.

[Jon80] C. B. Jones, *Software Development: A Rigorous Approach*, Prentice Hall International, 1980.

[Jon86] C. B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International, 1986.

[JPZ95] R. Janicki, D.L. Parnas, J. Zucker, “Tabular Representations in Relational Documents”, *CRL Report* 313, McMaster University, November 1995. Also in, Software Fundamentals - Collected Papers by David L. Parnas, edited by D. M. Hoffman and D. M. Weiss, Addison-Wesley, 2001, pp. 71-87.

[Kal90] A. Kaldewaij, *Programming: The Derivation of Algorithms*, Prentice Hall International, 1990.

[Kle98] T. Kleymann, “Hoare logic and auxiliary variables”, *Form Aspects of Computing*, Vol. 11, 1999, pp. 541-566.

[Kow77] T. Kowaltowski, “Axiomatic approach to side effects and general jumps”, *Acta Informatica*, Vol. 7, 1977, pp. 357-360.

[Kul02] G. Kulczycki, “Efficient Reusable Components with Value Semantics,” *Proceedings of the ICSR 2002 Young Researcher’s Workshop,* Austin TX, April 2002.

[Lig91] D. Lightfoot, *Formal Specification using Z*, Macmillan, 1991.

[LMW79] R. C. Linger, H. D. Mills, B. I. Witt, *Structured Programming - Theory and Practice*, Addison-Wesley publishing company, 1979.

[Luc87] P. Lucas, “VDM: Origins, Hopes, and Achievements”, *Proceedings of VDM’87, VDM-Europe Symposium 1987*, Brussels, Belgium, March 1987.

[Mad75] R. A. Maddus, *A Study of Computer Program Structure*, PhD. dissertation, University of Waterloo, Canada, 1975.

[Maj80] M. E. Majster-Cederbaum, “A simple relation between relational and predicate transformer semantics for nondeterministic program”, *Information Processing Letters*, Vol. 11, 1980.

[Man69] Z. Manna, “The correctness of program.” *Journal of Computer and Systems Sciences*, 1969, pp. 119-127.

[MBG87] H. D. Mills, V. R. Basili, J. D. Gannon and R. G. Hamlet, *Principles of Computer Programming: A Mathematical Approach*, William C. Brown, Dubuque, IA, 1987.

[MBG89] H. D. Mills, V. R. Basili, J. D. Gannon, “Mathematical Principles for a First Course in Software Engineering”, *IEEE Trans. on Software Engineering*, Vol. 15, No. 5, 1989, pp. 550-559.

[McC62] J. McCarthy, “Towards a Mathematical Science of Computation”, *Information Processing*, North-Holland, 1962, pp. 21-28

[McC63] J. McCarthy, “A Basis for a Mathematical Theory of Computation”, *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg, North-Holland, Amsterdam, 1963, pp. 33-70.

[Mil75a] H. D. Mills, “The New Math of Computer Programming”, *Communications of the ACM*, Vol. 18, No. 1, January 1975, pp. 43-48.

[Mil75b] H. D. Mills, “How to write correct programs and know it”, *Proceedings of the international conference on Reliable software*, Los Angeles, California, April 1975, pp. 363-370.

[Mil88] H. D. Mills, “Stepwise Refinement and Verification in Box-Structured Systems”, *IEEE Computer*, Vol. 21, No. 6, June 1988, pp. 23-36.

[MK75] C. L. McGowan, J. R. Kelly, *Top-down Structured Programming Techniques*, Petrocelli/Charter, New York, 1975.

[Mor90] C. Morgan, *Programming from Specifications*, Prentice Hall International, 1990.

[MR67] A. R. Meyer and D. M. Ritchie, “The Complexity of Loop Programs”, *Proc. 22nd National ACM Conf.*, Thompson, Washington, D.C., August 1967, pp. 465-470.

[Nau66] P. Naur, “Proofs of algorithms by General Snapshots”, *BIT*, Vol. 6, 1966, pp. 310-316.

[PAM91] D. L. Parnas, G. J. K. Asmis, J. Madey, “Assessment of Safety-Critical Software in Nuclear Power Plants”, *Nuclear Safety*, Vol. 32, No. 2, April-June 1991, pp. 189-198.

[Par72] D. L. Parnas, “On the Criteria To Be Used in Decomposing Systems into Modules”, *Communications of the ACM*, Vol. 18, No. 12, 1972, pp. 1053-1058.

[Par83] D. L. Parnas, "A Generalized Control Structure and Its Formal Definition", *Communications of the ACM*, Vol. 26, No. 8, August 1983, pp. 572-581.

[Par86] D. L. Parnas, P. C. Clements, “A Rational Design Process: How and Why to Fake it”, *IEEE Trans. On Software Engineering*, Vol. SE-12, No. 2, February 1986.

[Par92] D. L. Parnas, “Tabular representation of relations”, *CRL Report* 260, Telecommunications Research Institute of Ontario (TRIO), McMaster University, October 1992.

[Par93a] D. L. Parnas, "Some Theorems We Should Prove", *Proceedings of 1993 International Meeting on Higher Order Logic Theorem Proving and Its Applications,* The University of British Columbia, Vancouver, BC, August 10 - 13, 1993, pp. 156 - 163.

[Par93b] D. L. Parnas, “Mathematics of Computation for (Software and Other) Engineers”, *Bulletin of the European Association for Theoretical Computer Science,* No. 51, October 1993, pp. 249-259.

- Also in *Proceedings of the Third International Conference on Algebraic Methodology and Software Technology,* University of Twente, Netherlands, June 21-25, 1993.
- Also in *Proceedings of the First IMA Conference on Mathematics of Dependable Systems,* University of London, England, September 1993.
- Also in *Mathematics of Dependable Systems,* edited by C. Mitchell and V. Stavridou, Claredon Press, Oxford, 1995, pp. 209-224 (small revisions).

[Par94a] D. L. Parnas, "Mathematical Descriptions and Specification of Software", *Proceedings of IFIP World Congress 1994,* Vol. I, August 1994, pp. 354 - 359.

[Par94b] D. L. Parnas, “Inspection of Critical Software using Function Tables”, *Proceedings of IFIP World Congress 1994,* Vol. III, August 1994, pp. 270 - 277.

[Par95] D.L. Parnas, "A Logic for Describing, not Verifying, Software", *Erkenntnis* (Kluwer), Vol. 43, No. 3, November 1995, pp. 321-338.

[Par96] D.L. Parnas, “Mathematical methods: what we need and don’t need”, *IEEE Computer*, Vol. 29, No. 4, April, 1996, pp. 28-29. (In roundtable "An Invitation to Formal Methods").

[Par97] D. L. Parnas, “Precise description and specification of software”, *Mathematics of Dependable Systems II*, edited by V. Stavridou, Clarendon Press, 1997, pp. 1-14.

[Par98] D. L. Parnas, “ ‘Formal Methods’ technology transfer will fail”, *J. Systems Software*, Vol. 40, 1998, pp. 195-198.

[PBD85] J. S. Pedersen D. Bjørner, T. Denvir, E. Meiling, “The RAISE Project Fundamental Issues and Requirements”, Technical Report, Dansk Datamatik Center, Number RAISE/DDC/EM/1/V6, 1985.

[Pet96] D. K. Peters, “Shortest Path Algorithm: Formal Program Documentation”, *CRL Report* 280 (draft), Telecommunications Research Institute of Ontario (TRIO), McMaster University, February 1996.

[PL98] D. L. Parnas, A. Lawton, “Precisely Annotated Hierarchical Pictures of Programs”, *CRL Report* 359, 1998.

[PL03] D. L. Parnas, M. Lawford, “The Role of Inspection in Software Quality Assurance”, *IEEE Trans. on Software Engineering*, Vol. 29, No. 8, August 2003, pp. 674-676.

[PMI94] D. L., Parnas, J. Madey, M. Iglewski, "Precise Documentation of Well-Structured Programs", *IEEE Trans. on Software Engineering*, Vol. 20, No.12, December 1994, pp. 948 - 976.

[PST91] B. F. Potter, J. E. Sinclair, D. Till, *An Introduction to Formal Specification and Z*, Prentice-Hall, first edition, 1991.

[PW85] D. L. Parnas, D. M. Weiss, “Active Design Reviews: Principles and Practices”, *Proceedings of Eighth International Conference on Software Engineering*, August 1985, pp. 132-136. Also in, *Software Fundamentals - Collected Papers by David L. Parnas*, edited by D. M. Hoffman and D. M. Weiss, Addison-Wesley, 2001, pp. 339-354.

[PZ01] T. W. Pratt, M. V. Zelkowitz, *Programming Languages Design and Implementation*, Fourth Edition, Prentice Hall, 2001.

[RTI02] Prepared by RTI for the National Institute of Standards and Technology (NIST), “The Economic Impacts of Inadequate Infrastructure for Software Testing”. *Planning Report 02-3*, May 2002.

[Sch99] G. Schellhorn, *Verification of Abstract State Machine*, Doctoral Thesis of computer science at the University of Ulm, 1999.

[Sch01] S. Schneider, *The B-Method: An Introduction*, Macmillan Publishers Ltd, Houndmills, Basingstoke, Hampshire, England, October 2001.

[SERG97]McMaster University Software Engineering Research Group, “Table Tool System Developer's Guide”, *CRL Report* 339, Telecommunications Research Institute of Ontario (TRIO), McMaster University, January 1997. http://www.cas.mcmaster.ca/serg/ crl_reports.html (December 2004).

[Spi89] J. M. Spivey, *The Z Notation: a reference manual*, Prentice-Hall, Hemel Hempstead, 1989.

[Tur49] A. M. Turing, "Checking a large routine", *Report of a Conference on High Speed Automatic Calculating Machines*, University Mathematical Laboratory, Cambridge, June 1949, pp. 67-69.

[Wir73] N. Wirth, *Systematic Programming - An Introduction*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

Appendix A

Program Documentation – Specification, Design and Verification for the Partition Subprogram

A.1 Introduction

This appendix presents the documentation, design process and verification for a subprogram – Partition, which rearranges an array in a specific order. This example was taken from [Bab87]. The documentation structure uses Parnas' Display method [PMI94].

The documentation for each implementation is presented as a set of displays, supplemented by a lexicon and an index. Instead of using a relation in the display presented in [PMI94], the pre/postcondition pair is used in order to connect with the design and verification process which adopts Baber's approach.

Each display consists of five parts:

Part 1: a specification for the program presented in this display,
Part 2: the justification and illustration of some major design decisions,
Part 3: the program itself,
Part 4: specifications of all programs invoked in Part 3,
Part 5: the demonstration of the correctness of the program. This could be either an informal reasoning or a formal verification. For some simple programs, direct assertions can be used.

A.2 Documentation of Partition subprogram

A.2.1 Informal description

The program will re-arrange an array into three regions – the lower, middle and upper regions – such that the middle region may not be empty. These three regions will respectively contain array elements whose values are less than, equal to and greater than some initially selected value.

Some conventions are used. We use different fonts to distinguish between programming language elements (e.g. “Partition”), and mathematical terms (e.g. “*Perm*”). ‘X is a specification variable that represents the initial value of X.

A.2.2 Documentation

DISPLAY 1

Part 1: Display 1 Specification

Partition	external variable: X, n, left, right
Precondition (V)	$n\in\mathbb{Z} \wedge \mathbb{Z}\subseteq\text{Set.“left”} \wedge \mathbb{Z}\subseteq\text{Set.“right”} \wedge\ 1\le n\ \wedge_{i=1}^{n} X[i]\in R$ $\wedge_{i=1}^{n} X[i]=‘X[i]$
Postcondition (P)	$n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le\text{left}\le\text{right}\le n$ $\wedge_{i=1}^{\text{left-1}} X[i]<X[\text{left}] \wedge_{i=\text{left}}^{\text{right}} X[i]=X[\text{left}] \wedge_{i=\text{right}+1}^{n} X[i]>X[\text{left}]$ $\wedge(\&_{i=1}^{n} X[i])$ *Perm* $(\&_{i=1}^{n}\ ‘X[i])$

For every data environment (defined in 3.4, 3.7) d satisfying the precondition V:
Partition.d = d except for the values of left, right and X[i], i=1, …n.

Part 2: Display 1 Design

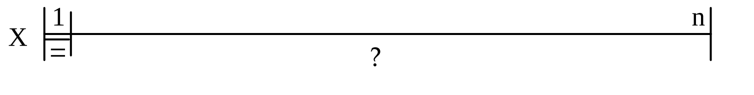
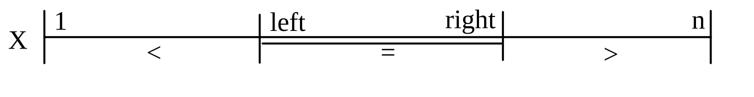
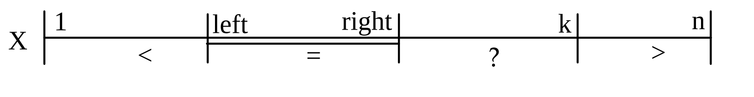
1. Basic structure

The permutation of the array involves the repetition of exchanging the values of the array, which suggests a loop as the basic structure for this program.

2. Loop invariant I

A loop invariant is the generalization of the initial and final status of the loop execution.

***Diagrams of the Loop Execution Status*:**

Initial status	
Final status	
Generalization (I)	

Legend:
- = meaning equal to each other,
- < meaning that each element is less than the elements in the "=" region,
- > meaning that each element is greater than the elements in the "=" region.
- A double line represents the "=" region, which may not be empty. In the initial state, "=" region could be anywhere in the array.

I: $n\in\mathbb{Z} \wedge left\in\mathbb{Z} \wedge right\in\mathbb{Z} \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\leq left\leq right\leq k\leq n$

$\wedge_{i=1}^{left-1} X[i]<X[left] \wedge_{i=left}^{right} X[i]=X[left] \wedge_{i=k+1}^{n} X[i]>X[left]$

$\wedge(\&_{i=1}^{n}$ X[i]) *Perm* $(\&_{i=1}^{n}$ 'X[i])

3. Initialization

The initialization establishes the truth of I. Since the value of an element in "=" region can be arbitrarily, the initialization could be:

```
left = 1;                    /* < region empty */
right = left;                /* = region has one element */
int    k = n;                /* > region empty */
```

Optionally, we may explicitly choose the value in "=" region by adding:
swap(X[left], X[j]), where $j \in \mathbb{Z} \wedge 1 \le j \le n$

4. Loop Condition B

I and P differ only by the one term

I: $\wedge_{i=k+1}^{n} X[i]$

P: $\wedge_{i=right+1}^{n} X[i]$

By $I \Rightarrow (\neg B \Rightarrow P)$ and assuming the truth of I, then we have $(right=k) \Rightarrow P$. Hence, $right \neq k$ is a suitable candidate for B.

Given the truth of I, $right \neq k$ is equivalent to the condition $right<k$, because the latter is stronger, we select it, although either condition is perfectly suitable.

5. Loop Body S

We design loop body S according to the guidelines:

(1) maintains the truth of I,
(2) makes progress toward fulfilling the postcondition, i.e. loop termination.

According to the diagram of I, the ? region must be reduced to meet (2). Then the state corresponding to I must be re-established to meet (1). One of the possible solutions is to start from X[right+1], evaluate it and then insert it to one of the three regions. Alternatively, one may start from X[k] instead. We will use the former solution here.

Case analysis for S

	X[right+1] < X[left]	X[right+1] = X[left]	X[right+1] > X[left]
Reduce ? region	swap(X[left], X[right+1])	Already in = region	swap(X[right+1], X[k])

Re-establish the truth of I	left := left +1
	right := right+1
		right := right+1	k := k-1

The above analysis suggests the if statement for S.

Part 3: Display 1 Program

```
/*  This program is implemented in C.  */
    void partition (void)
    {
            left = 1;
            right = left;
            int   k = n;

            while ( right < k )
            {
                    if ( X[right+1] < X[left] )
                    {
                            swap( X[left], X[right+1]);
                            left = left+1;
                            right = right+1;
                    }
                    else if ( X[right+1] == X[left] )
                            right = right+1;
                         else
                         {
                            swap( X[right+1], X[k]);
                            k = k-1;
                          }
            }
    }   /* This last } implicitly includes the statement "release k". */
```

Part 4: Display 1 Specifications of Invoked Programs

swap (a, b)	External variables: none
Precondition (V)	a∈R ∧ b∈R ∧ a = ‘a ∧ b = ‘b
Postcondition (P)	a = ‘b ∧ b = ‘a

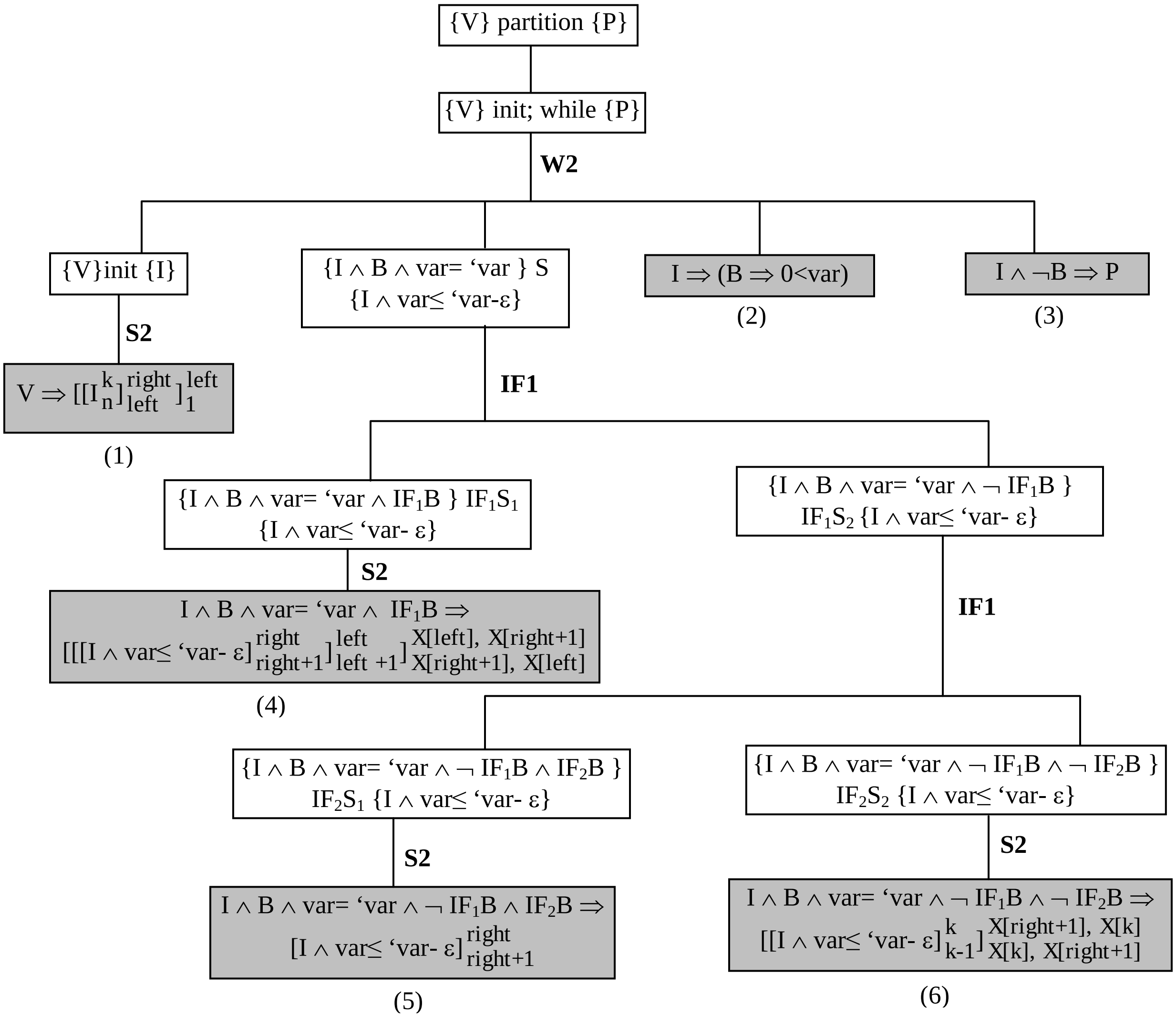
(*on Display 2*)

For all data environment d satisfying the precondition V:
swap.d = d except for the values of the variables *a* and *b*.

Part 5: Display 1 Program Correctness

The theorem “…Partition.d = d except for the values of left, right and X[i], i=1, …n” can be verified informally by inspecting the program segment. The correctness proposition {V} S {P} will be first decomposed into the Boolean algebraic expressions. Then the proof will be presented.

1. Decomposition of the correctness proposition


{V} partition {P}
{V} init; while {P}
W2
{V}init {I}
{I ∧ B ∧ var= ‘var } S {I ∧ var≤ ‘var-ε}
I ⇒ (B ⇒ 0<var)
(2)
I ∧ ¬B ⇒ P
(3)
S2
$V \Rightarrow [[I^{k}_{n}]^{right}_{left}]^{left}_{1}$
(1)
IF1
$\{I \wedge B \wedge var= `var \wedge IF_1B\} IF_1S_1 \{I \wedge var\leq `var- \varepsilon\}$
$\{I \wedge B \wedge var= `var \wedge \neg IF_1B\} IF_1S_2 \{I \wedge var\leq `var- \varepsilon\}$
S2
$I \wedge B \wedge var= `var \wedge IF_1B \Rightarrow [[[I \wedge var\leq `var- \varepsilon]^{right}_{right+1}]^{left}_{left +1}]^{X[left], X[right+1]}_{X[right+1], X[left]}$
(4)
IF1
$\{I \wedge B \wedge var= `var \wedge \neg IF_1B \wedge IF_2B\} IF_2S_1 \{I \wedge var\leq `var- \varepsilon\}$
$\{I \wedge B \wedge var= `var \wedge \neg IF_1B \wedge \neg IF_2B\} IF_2S_2 \{I \wedge var\leq `var- \varepsilon\}$
S2
$I \wedge B \wedge var= `var \wedge \neg IF_1B \wedge IF_2B \Rightarrow [I \wedge var\leq `var- \varepsilon]^{right}_{right+1}$
(5)
S2
$I \wedge B \wedge var= `var \wedge \neg IF_1B \wedge \neg IF_2B \Rightarrow [[I \wedge var\leq `var- \varepsilon]^{k}_{k-1}]^{X[right+1], X[k]}_{X[k], X[right+1]}$
(6)

Legend:

- IF_i, $i = 1, 2, \ldots$ denotes the *i*th if statement encountered in the program,
- IF_iB, $i = 1, 2, \ldots$ denotes the condition of the *i*th if statement encountered in the program,
- S_j, $j \in \{1, 2\}$ denotes the body of (1) then part or (2) else part of an if statement,
- var denotes the loop variant function,
- **S2** denotes the rule for a sequence of assignment statements,
- **W2** denotes the rule for the initialized while loop,
- **IF1** denotes the rule for the if statement.
- ▨ contains the final decomposed Boolean expressions to be verified.

2. Proof[1]

We choose *k-right* as *var* so that B=(0<*var*) (where B is *right*<*k*). Each execution of the loop body either increases *right* by 1 or decreases *k* by 1, and hence always decreases *var* by 1. Therefore we choose ε to be 1 so that the other part of the proof of termination can be completed.

(1) is true:

$$[[I^{k}_{n}]^{right}_{left}]^{left}_{1}$$

=

$$n\in\mathbb{Z} \wedge 1\in\mathbb{Z} \wedge 1\in\mathbb{Z} \wedge n\in\mathbb{Z} \wedge^{n}_{i=1} X[i]\in R \wedge 1\leq n$$

$$\wedge^{0}_{i=1} X[i]<X[1] \wedge^{1}_{i=1} X[i]=X[1] \wedge^{n}_{i=n+1} X[i]>X[1]$$

$$\wedge(\&^{n}_{i=1}\ X[i])\ \mathit{Perm}\ (\&^{n}_{i=1}\ `X[i])$$

$\Leftarrow$

$$n\in\mathbb{Z} \wedge \mathbb{Z}\subseteq Set.\text{“left”} \wedge \mathbb{Z}\subseteq Set.\text{“right”} \wedge 1\leq n\ \wedge^{n}_{i=1} X[i]\in R \wedge^{n}_{i=1} X[i]=`X[i]$$

=

$$V\ \blacksquare$$

(2) is true:

$$I \Rightarrow (B \Rightarrow 0<var)$$

=

$$I \Rightarrow (right<k \Rightarrow 0<k\text{-}right)$$

=

$$I \Rightarrow true$$

=

$$true\ \blacksquare$$

[1] Note that semiformal versions of such proofs can be illustrated with diagrams like the one used for I. They are less formal and less rigorous, but can be constructed and reviewed much more quickly and easily. An example is on pages 290 and 291 of [Bab87]. Such a semiformal version of a proof is very much like the design steps we went through at the beginning of this section of the Appendix.

(3) is true:

$$I \wedge \neg B$$

=

$$n \in \mathbb{Z} \wedge left \in \mathbb{Z} \wedge right \in \mathbb{Z} \;\wedge k \in \mathbb{Z} \wedge_{i=1}^{n} X[i] \in R \wedge 1 \leq left \leq right = k \leq n$$

$$\wedge_{i=1}^{left\text{-}1} X[i] < X[left] \wedge_{i=left}^{right} X[i] = X[left] \wedge_{i=k+1}^{n} X[i] > X[left]$$

$$\wedge (\&_{i=1}^{n}\, X[i])\ \textit{Perm}\ (\&_{i=1}^{n}\, \text{‘}X[i])$$

⇒

$$n \in \mathbb{Z} \wedge left \in \mathbb{Z} \wedge right \in \mathbb{Z} \wedge_{i=1}^{n} X[i] \in R \wedge 1 \leq left \leq right \leq n$$

$$\wedge_{i=1}^{left\text{-}1} X[i] < X[left] \wedge_{i=left}^{right} X[i] = X[left] \wedge_{i=right+1}^{n} X[i] > X[left]$$

$$\wedge (\&_{i=1}^{n}\, X[i])\ \textit{Perm}\ (\&_{i=1}^{n}\, \text{‘}X[i])$$

=

$$P \;\blacksquare$$

(4) is true:

$$[[[n \in \mathbb{Z} \wedge left \in \mathbb{Z} \wedge right \in \mathbb{Z} \;\wedge k \in \mathbb{Z} \wedge_{i=1}^{n} X[i] \in R \wedge 1 \leq left \leq right \leq k \leq n$$

$$\wedge_{i=1}^{left\text{-}1} X[i] < X[left] \wedge_{i=left}^{right} X[i] = X[left] \wedge_{i=k+1}^{n} X[i] > X[left]$$

$$\wedge (\&_{i=1}^{n}\, X[i])\ \textit{Perm}\ (\&_{i=1}^{n}\, \text{‘}X[i])$$

$$\wedge\ k\text{-}right \leq (\text{‘}k\text{- ‘}right)\text{-}1]_{right+1}^{right}]_{left\ +1}^{left}]_{X[right+1],\ X[left]}^{X[left],\ X[right+1]}$$

=

$$[[[n \in \mathbb{Z} \wedge left \in \mathbb{Z} \wedge right \in \mathbb{Z} \;\wedge k \in \mathbb{Z} \wedge_{i=1}^{n} X[i] \in R \wedge 1 \leq left \leq right \leq k \leq n$$

$$\wedge_{i=1}^{left\text{-}1} X[i] < X[left] \wedge_{i=left}^{right} X[i] = X[left] \wedge_{i=k+1}^{n} X[i] > X[left]$$

$$\wedge (\&_{i=1}^{right\text{-}1}\, X[i]\ \& X[right]\ \&_{i=right+1}^{n}\, X[i])$$

$$\textit{Perm}\ (\&_{i=1}^{right\text{-}1}\text{‘}X[i]\ \&\text{‘}X[right]\ \&_{i=right+1}^{n}\text{‘}X[i])$$

$$\wedge\ k\text{-}right \leq (\text{‘}k\text{- ‘}right)\text{-}1]_{right+1}^{right}]_{left\ +1}^{left}]_{X[right+1],\ X[left]}^{X[left],\ X[right+1]}$$

=

$[[n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le\text{left}\le\text{right}+1\le k\le n$

$\wedge_{i=1}^{\text{left}-1} X[i]<X[\text{left}] \wedge_{i=\text{left}}^{\text{right}+1} X[i]=X[\text{left}] \wedge_{i=k+1}^{n} X[i]>X[\text{left}]$

$\wedge(\&_{i=1}^{\text{right}} X[i] \ \& X[\text{right}+1] \ \&_{i=\text{right}+2}^{n} X[i])$

$\quad \mathit{Perm}\ (\&_{i=1}^{\text{right}}\, `X[i] \ \&`X[\text{right}+1] \ \&_{i=\text{right}+2}^{n}\, `X[i])$

$\wedge\ k-(\text{right}+1)\le(`k-\ `\text{right})-1]\ {}^{\text{left}}_{\text{left}\ +1}]^{X[\text{left}],\ X[\text{right}+1]}_{X[\text{right}+1],\ X[\text{left}]}$

=

$[[n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le\text{left}\le\text{right}+1\le k\le n$

$\wedge_{i=1}^{\text{left}-1} X[i]<X[\text{left}] \wedge_{i=\text{left}}^{\text{right}+1} X[i]=X[\text{left}] \wedge_{i=k+1}^{n} X[i]>X[\text{left}]$

$\wedge(\&_{i=1}^{n} X[i])\ \mathit{Perm}\ (\&_{i=1}^{n}\ `X[i])$

$\wedge\ k-(\text{right}+1)\le(`k-\ `\text{right})-1]\ {}^{\text{left}}_{\text{left}\ +1}]^{X[\text{left}],\ X[\text{right}+1]}_{X[\text{right}+1],\ X[\text{left}]}$

=

$[n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le\text{left}+1\le\text{right}+1\le k\le n$

$\wedge_{i=1}^{\text{left}} X[i]<X[\text{left}+1]$

$\wedge_{i=\text{left}+1}^{\text{right}+1} X[i]=X[\text{left}+1]$

$\wedge_{i=k+1}^{n} X[i]>X[\text{left}+1]$

$\wedge(\&_{i=1}^{n} X[i])\ \mathit{Perm}\ (\&_{i=1}^{n}\ `X[i]) \wedge k-\text{right}\le\ `k-\ `\text{right}]\ ^{X[\text{left}],\ X[\text{right}+1]}_{X[\text{right}+1],\ X[\text{left}]}$

=

$[n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le\text{left}+1\le\text{right}+1\le k\le n$

$\wedge_{i=1}^{\text{left}} X[i]<X[\text{left}+1]$

$\wedge\ {}_{i=\text{left}+1}^{\text{right}} X[i]=X[\text{left}+1] \wedge X[\text{right}+1]=X[\text{left}+1]$

$\wedge_{i=k+1}^{n} X[i]>X[\text{left}+1]$

$\wedge(\&_{i=1}^{n} X[i])\ \text{Perm}\ (\&_{i=1}^{n}\ `X[i]) \wedge k-\text{right}\le\ `k-\ `\text{right}]\ ^{X[\text{left}],\ X[\text{right}+1]}_{X[\text{right}+1],\ X[\text{left}]}$

=

$[n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le\text{left}+1\le\text{right}+1\le k\le n$

$\wedge_{i=1}^{\text{left}} X[i]<X[\text{right}+1]$

$\wedge_{i=left+1}^{right} X[i]=X[right+1] \wedge X[right+1]=X[left+1]$

$\wedge_{i=k+1}^{n} X[i]>X[right+1]$

$\wedge(\&_{i=1}^{n}\ X[i])\ \text{Perm}\ (\&_{i=1}^{n}\ ‘X[i]) \wedge k\text{-}right \leq ‘k\text{-}\ ‘right\ {}^{X[left],\ X[right+1]}_{X[right+1],\ X[left]}$

=

$[n\in\mathbb{Z} \wedge left\in\mathbb{Z} \wedge right\in\mathbb{Z}\ \ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\leq left+1\leq right+1\leq k\leq n$

$\wedge_{i=1}^{left-1} X[i]<X[right+1] \wedge X[left]<X[right+1]$

$\wedge_{i=left+1}^{right} X[i]=X[right+1]$

$\wedge_{i=k+1}^{n} X[i]>X[right+1]$

$\wedge(\&_{i=1}^{n}\ X[i])\ \text{Perm}\ (\&_{i=1}^{n}\ ‘X[i]) \wedge k\text{-}right \leq ‘k\text{-}\ ‘right]\ {}^{X[left],\ X[right+1]}_{X[right+1],\ X[left]}$

=

[this substitution exchanges two elements of the array, so *perm* will not change]

$n\in\mathbb{Z} \wedge left\in\mathbb{Z} \wedge right\in\mathbb{Z}\ \ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\leq left+1\leq right+1\leq k\leq n$

$\wedge_{i=1}^{left-1} X[i]<X[left] \wedge X[right+1]<X[left]$

$\wedge_{i=left+1}^{right} X[i]=X[left]$

$\wedge_{i=k+1}^{n} X[i]>X[left]$

$\wedge(\&_{i=1}^{n}\ X[i])\ \text{Perm}\ (\&_{i=1}^{n}\ ‘X[i]) \wedge k\text{-}right \leq ‘k\text{-}\ ‘right$

$\Leftarrow$

$n\in\mathbb{Z} \wedge left\in\mathbb{Z} \wedge right\in\mathbb{Z}\ \ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\leq left\leq right<k\leq n$

$\wedge_{i=1}^{left-1} X[i]<X[left] \wedge X[right+1]<X[left]$

$\wedge_{i=left}^{right} X[i]=X[left]$

$\wedge_{i=k+1}^{n} X[i]>X[left]$

$\wedge(\&_{i=1}^{n}\ X[i])\ \textit{Perm}\ (\&_{i=1}^{n}\ ‘X[i]) \wedge k\text{-}right= ‘k\text{-}\ ‘right$

=

$I \wedge right<k \wedge k\text{-}right= ‘k\text{-}\ ‘right \wedge X[right+1]<X[left]$ ■

(5) is true:

$$[I \wedge \text{var} \leq \text{‘var} - \varepsilon]^{\text{right}}_{\text{right+1}}$$

=

$$[n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \wedge k\in\mathbb{Z} \wedge^{n}_{i=1} X[i]\in R \wedge 1\leq\text{left}\leq\text{right}\leq k\leq n$$

$$\wedge^{\text{left-1}}_{i=1} X[i]<X[\text{left}] \wedge^{\text{right}}_{i=\text{left}} X[i]=X[\text{left}] \wedge^{n}_{i=k+1} X[i]>X[\text{left}]$$

$$\wedge(\&^{n}_{i=1} X[i]) \mathit{Perm} (\&^{n}_{i=1} \text{‘}X[i])$$

$$\wedge\ k\text{-right}\leq(\text{‘}k\text{- ‘right})\text{-}1]^{\text{right}}_{\text{right+1}}$$

=

$$n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \wedge k\in\mathbb{Z} \wedge^{n}_{i=1} X[i]\in R \wedge 1\leq\text{left}\leq\text{right}+1\leq k\leq n$$

$$\wedge^{\text{left-1}}_{i=1} X[i]<X[\text{left}] \wedge^{\text{right+1}}_{i=\text{left}} X[i]=X[\text{left}] \wedge^{n}_{i=k+1} X[i]>X[\text{left}]$$

$$\wedge(\&^{n}_{i=1} X[i]) \mathit{Perm} (\&^{n}_{i=1} \text{‘}X[i]) \wedge k\text{-(right+1)}\leq(\text{‘}k\text{- ‘right})\text{-}1$$

$\Leftarrow$

$$n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \wedge k\in\mathbb{Z} \wedge^{n}_{i=1} X[i]\in R \wedge 1\leq\text{left}\leq\text{right}<k\leq n$$

$$\wedge^{\text{left-1}}_{i=1} X[i]<X[\text{left}] \wedge^{\text{right}}_{i=\text{left}} X[i]=X[\text{left}] \wedge X[\text{right}+1]=X[\text{left}]$$

$$\wedge^{n}_{i=k+1} X[i]>X[\text{left}]$$

$$\wedge(\&^{n}_{i=1} X[i]) \mathit{Perm} (\&^{n}_{i=1} \text{‘}X[i]) \wedge k\text{-right}=\text{‘}k\text{- ‘right}$$

=

$$I \wedge \text{right}<k \wedge k\text{-right}=\text{‘}k\text{- ‘right} \wedge X[\text{right}+1]=X[\text{left}]$$

=

$$I \wedge \text{right}<k \wedge k\text{-right}=\text{‘}k\text{- ‘right} \wedge X[\text{right}+1]\geq X[\text{left}] \wedge X[\text{right}+1]=X[\text{left}] \ \blacksquare$$

(6) is true:

$$[[I \wedge \text{var}\leq \text{‘var} - \varepsilon]^{k}_{k-1}]^{X[\text{right}+1],\ X[k]}_{X[k],\ X[\text{right}+1]}$$

=

$$[n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \wedge k\in\mathbb{Z} \wedge^{n}_{i=1} X[i]\in R \wedge 1\leq\text{left}\leq\text{right}\leq k\leq n$$

$$\wedge^{\text{left-1}}_{i=1} X[i]<X[\text{left}] \wedge^{\text{right}}_{i=\text{left}} X[i]=X[\text{left}] \wedge^{n}_{i=k+1} X[i]>X[\text{left}]$$

$$\wedge(\&^{n}_{i=1} X[i]) \mathit{Perm} (\&^{n}_{i=1} \text{‘}X[i])$$

$$\wedge\ k\text{-right}\leq(\text{‘}k\text{- ‘right})\text{-}1]^{k}_{k-1}]^{X[\text{right}+1],\ X[k]}_{X[k],\ X[\text{right}+1]}$$

$=$

$$\begin{aligned}
&[n\in\mathbb{Z} \wedge \mathit{left}\in\mathbb{Z} \wedge \mathit{right}\in\mathbb{Z}\ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le \mathit{left}\le \mathit{right}\le k\text{-}1\le n\\
&\wedge_{i=1}^{\mathit{left}\text{-}1} X[i]<X[\mathit{left}] \wedge_{i=\mathit{left}}^{\mathit{right}} X[i]=X[\mathit{left}] \wedge_{i=k}^{n} X[i]>X[\mathit{left}]\\
&\wedge(\&_{i=1}^{n}\ X[i])\ \mathit{Perm}\ (\&_{i=1}^{n}\ \text{‘}X[i])\\
&\wedge\ k\text{-}\mathit{right}\le \text{‘}k\text{- ‘}\mathit{right}]^{X[\mathit{right}+1],\ X[k]}_{X[k],\ X[\mathit{right}+1]}
\end{aligned}$$

$\Leftarrow$

$$\begin{aligned}
&[n\in\mathbb{Z} \wedge \mathit{left}\in\mathbb{Z} \wedge \mathit{right}\in\mathbb{Z}\ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le \mathit{left}\le \mathit{right}\le k\le n\\
&\wedge_{i=1}^{\mathit{left}\text{-}1} X[i]<X[\mathit{left}] \wedge_{i=\mathit{left}}^{\mathit{right}} X[i]=X[\mathit{left}] \wedge_{i=k}^{n} X[i]>X[\mathit{left}]\\
&\wedge(\&_{i=1}^{n}\ X[i])\ \mathit{Perm}\ (\&_{i=1}^{n}\ \text{‘}X[i])\\
&\wedge\ k\text{-}\mathit{right}\le \text{‘}k\text{- ‘}\mathit{right}]^{X[\mathit{right}+1],\ X[k]}_{X[k],\ X[\mathit{right}+1]}
\end{aligned}$$

$=$

$$\begin{aligned}
&[n\in\mathbb{Z} \wedge \mathit{left}\in\mathbb{Z} \wedge \mathit{right}\in\mathbb{Z}\ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le \mathit{left}\le \mathit{right}\le k\le n\\
&\wedge_{i=1}^{\mathit{left}\text{-}1} X[i]<X[\mathit{left}] \wedge_{i=\mathit{left}}^{\mathit{right}} X[i]=X[\mathit{left}]\\
&\wedge_{i=k+1}^{n} X[i]>X[\mathit{left}] \wedge X[k]>X[\mathit{left}]\\
&\wedge(\&_{i=1}^{n}\ X[i])\ \mathit{Perm}\ (\&_{i=1}^{n}\ \text{‘}X[i])\\
&\wedge\ k\text{-}\mathit{right}\le \text{‘}k\text{- ‘}\mathit{right}]^{X[\mathit{right}+1],\ X[k]}_{X[k],\ X[\mathit{right}+1]}
\end{aligned}$$

$=$

$$\begin{aligned}
&n\in\mathbb{Z} \wedge \mathit{left}\in\mathbb{Z} \wedge \mathit{right}\in\mathbb{Z}\ \wedge k\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le \mathit{left}\le \mathit{right}\le k\le n\\
&\wedge_{i=1}^{\mathit{left}\text{-}1} X[i]<X[\mathit{left}] \wedge_{i=\mathit{left}}^{\mathit{right}} X[i]=X[\mathit{left}]\\
&\wedge_{i=k+1}^{n} X[i]>X[\mathit{left}] \wedge X[\mathit{right}+1]>X[\mathit{left}]\\
&\wedge(\&_{i=1}^{n}\ X[i])\ \mathit{Perm}\ (\&_{i=1}^{n}\ \text{‘}X[i]) \wedge k\text{-}\mathit{right}\le \text{‘}k\text{- ‘}\mathit{right}
\end{aligned}$$

$\Leftarrow$

$$I \wedge \mathit{right}<k \wedge k\text{-}\mathit{right}=\text{‘}k\text{- ‘}\mathit{right}\wedge X[\mathit{right}+1]>X[\mathit{left}]$$

$=$

$$I \wedge \mathit{right}<k \wedge k\text{-}\mathit{right}=\text{‘}k\text{- ‘}\mathit{right}\wedge X[\mathit{right}+1]\ge X[\mathit{left}] \wedge X[\mathit{right}+1]\ne X[\mathit{left}]\ ■$$

END OF DISPLAY 1

DISPLAY 2

Part 1: Display 2 Specification

swap (a, b)	External variables: none
Precondition (V)	a∈R ∧ b∈R ∧ a = ‘a ∧ b = ‘b
Postcondition (P)	a = ‘b ∧ b = ‘a

For all data environment d satisfying the precondition V:
swap.d = d except for the values of the variables *a* and *b*.

Part 2: Display 2 Design

Basic structure

Since the postcondition consists of subexpressions that build upon one another, a sequence of statements is appropriate. By using an intermediate “container” *temp*, the truth of the corresponding subexpression can be established by a sequence of assignment statements.

Part 3: Display 2 Program

```
void swap ( int a, int b )
{
        int temp;
        temp = a;
        a = b;
        b = temp;
}
```

Part 4: Display 2 Specification of Invoked Program

Empty

Part 5: Display 2 Program Correctness

The theorem “swap.d = d except for the values of the variables *a* and *b*” can be verified informally by inspection of the program segment. The correctness proposition {V} S {P} will be true by rule of sequence of assignment statement if $V \Rightarrow [[[P]^{b}_{temp}]^{a}_{b}]^{temp}_{a}$ holds.

Proof: $V \Rightarrow [[[P]^{b}_{temp}]^{a}_{b}]^{temp}_{a}$ is true:

$[[[a = \text{‘}b \wedge b = \text{‘}a]^{b}_{temp}]^{a}_{b}]^{temp}_{a}$

$=$

$[[a = \text{‘}b \wedge temp = \text{‘}a]^{a}_{b}]^{temp}_{a}$

$=$

$[b = \text{‘}b \wedge temp = \text{‘}a]^{temp}_{a}$

$=$

$b = \text{‘}b \wedge a = \text{‘}a$

$\Leftarrow$

V ■

END OF DISPLAY 2

LEXICON

A. Auxiliary functions

Perm: sequence × sequence → boolean
$(\&^{q}_{i=p}\ A[i])$ *Perm* $(\&^{q}_{i=p} B[i]) \overset{df}{=} \exists f\colon \{A[p]\ldots A[q]\} \leftrightarrow \{B[p]\ldots B[q]\}.\ \forall i,\ p \leq i \leq q.\ A[i] = B[f(i)]$, where $f\colon \{A[p]\ldots A[q]\} \leftrightarrow \{B[p]\ldots B[q]\}$means f is a bijection from A[i] to B[i].

B. C external definitions and declarations

```
# define n   /* n≥1 */
int  X[n+1];
int  left;
int  right;
```

INDEX

Name	**Used in**
A	$D2_3$
B	$D2_3$
k	$D1_{2, 3}$
key	$D1_2$
left	$D1_{1, 2, 3}$
n	$D1_{1, 2, 3\,,}\, L_B$
Perm	$D1_{1, 2}$, L_A
right	$D1_{1, 2, 3}$
swap	$D1_{2, 3, 4}$, $D2_{1, 3}$
temp	$D2_3$
X	$D1_{1, 2, 3, 4}$, L_B

Legend:

- D0 denotes the introduction,
- Di, i=1, 2… denotes Display i,
- Di_j, i=1, 2 …, j∈{1, 2, 3, 4} denotes Display i, part P_j,
- $Di_{j,k}$, i=1, 2 …, j, k∈{1, 2, 3, 4} denotes Display i, parts P_j and P_k,
- L_x, x=A,B denotes the lexicon, part x.

Appendix B

Program Specifications Written in Both Pre/postcondition and Relational Approaches

B.1 Introduction

This appendix presents program specifications using both pre/postcondition and relational approaches to illustrate their differences. Two examples will be presented. Each example will include an informal description and two versions of specifications – one uses Baber's method, the other uses Parnas' method. Unlike Appendix A, the presentation of each specification will be in their original format used in each approach.

Besides the conventions used in Appendix A, other conventions are used, which are specific to each approach:

In Baber's approach,

- the initial value of X is normally written X', not 'X, but that we use 'X here to be consistent in this appendix in order to avoid confusion,
- d represents a data environment. We discussed it in Section 3.4, 3.5 and 3.7.

In Parnas' approach,

- 'X and X' represent the initial and final values of X respectively.
- NC is a predicate symbol which means "Not Changed". It is defined in [IMP93] as follows: $NC(v_1, \ldots, v_m) \overset{df}{=} (v_1' = {}'v_1) \wedge \ldots \wedge (v_m' = {}'v_m)$, where $'v_i$ and v_i'

represent the initial and final values of v_i respectively. The values of variables mentioned nowhere in the specification are also not to be changed.

B.2 Specifications of Partition subprogram

B.2.1 Informal description

This is the same program as in Appendix A.

B.2.2 Specifications

- **Specification in Baber’s approach:**

{V} Partition {P} and for all d satisfying the precondition V: Partition.d = d except for the values of left, right and X[i], i=1, …n,
where V and P are defined to be

V: $n\in\mathbb{Z} \wedge \mathbb{Z}\subseteq \text{Set.“left”} \wedge \mathbb{Z}\subseteq \text{Set.“right”} \wedge 1\le n \ \wedge_{i=1}^{n} X[i]\in R \wedge_{i=1}^{n} X[i]=‘X[i]$

P: $n\in\mathbb{Z} \wedge \text{left}\in\mathbb{Z} \wedge \text{right}\in\mathbb{Z} \wedge_{i=1}^{n} X[i]\in R \wedge 1\le \text{left}\le \text{right}\le n$

$\wedge_{i=1}^{\text{left-1}} X[i]<X[\text{left}] \wedge_{i=\text{left}}^{\text{right}} X[i]=X[\text{left}] \wedge_{i=\text{right}+1}^{n} X[i]>X[\text{left}]$

$\wedge(\&_{i=1}^{n} X[i])$ *Perm* $(\&_{i=1}^{n}\ ‘X[i])$

- **Specification in Parnas’ approach:**

<table>
<tr><td><b>Partition</b></td><td>external variable: X, n, left, right</td></tr>
<tr><td colspan="2">$R_{\text{Partition(,)}} = 1\le n$
$\Rightarrow$
<i>ThreeRegion</i>(X’, left’, right’) $\wedge$ $(\&_{i=1}^{n} X[i]’)$ <i>Perm</i> $(\&_{i=1}^{n}\ ‘X[i])$</td></tr>
</table>

LEXICON

A. Auxiliary functions

Perm: sequence × sequence → boolean
$(\&_{i=p}^{q} A[i])$ *Perm* $(\&_{i=p}^{q} B[i]) \overset{df}{=} \exists f\!: \{A[p]\ldots A[q]\} \leftrightarrow \{B[p]\ldots B[q]\}.\ \forall i,\ p\leq i\leq q.\ A[i] = B[f(i)]$, where f: $\{A[p]\ldots A[q]\} \leftrightarrow \{B[p]\ldots B[q]\}$means f is a bijection from A[i] to B[i].

ThreeRegion: array × integer × integer → boolean
$$\begin{aligned} \mathit{ThreeRegion}(X, \mathit{left}, \mathit{right}) \overset{df}{=} 1\leq \mathit{left}\leq \mathit{right}\leq n \wedge \forall i(1\leq i\leq n)\ [&1\leq i<\mathit{left} \Rightarrow X[i]<X[\mathit{left}] \\ &\wedge \mathit{left}\leq i\leq \mathit{right} \Rightarrow X[i]=X[\mathit{left}] \\ &\wedge \mathit{right}<i\leq n \Rightarrow X[i]>X[\mathit{left}]\ ] \end{aligned}$$

Note that this auxiliary mathematical function could also have been introduced and used in Appendix A.

B. C external definitions and declarations

```
# define n   /* n≥1 */
int  X[n+1];
int  left;
int  right;
```

B.3 Specifications of ExtractStr subprogram

B.3.1 Informal description

This example is taken from course materials of [Bab02]. Subprogram ExtractStr cuts a given string and returns the remaining string. Its input are the string variable S and two integer variables p1 and p2. Its output is the value of the string variable RemnStr. p1 is the position where the cut starts in S and p2 is the length of the remaining string. RemnStr is an external variable.

B.3.2 Specifications

- **Specification in Baber's approach:**

{V} call ExtractStr {P} strictly and for all data environments d satisfying the strict precondition V: ExtractStr.d = d except for the value of RemnStr, where V and P are as follows:

V: $\mathrm{Str}\in \mathrm{string} \wedge \mathrm{p1}\in\mathbb{Z} \wedge \mathrm{p2}\in\mathbb{Z} \wedge \mathrm{string} \subseteq \mathrm{Set}.\text{“RemnStr”}$

P: $$\begin{array}{l} \mathrm{RemnStr} = \mathit{substring}(\ \mathrm{Str}, \\ \qquad \mathit{lim}\ (\ 1, \mathrm{p1}, \mathit{length}(\mathrm{Str})+1\ ), \\ \qquad \mathit{lim}\ (\ 0, \mathrm{p2}, \mathit{length}(\mathrm{Str})-\mathit{lim}(1, \mathrm{p1}, \mathit{length}(\mathrm{Str})+1) + 1)\ ) \\ \quad) \end{array}$$

- **Specification in Parnas' approach:**

ExtractStr (Str, p1, p2)	external variable: RemnStr

$R_{\mathrm{ExtractStr}(,)}$ = NC(Str, p1, p2) ∧ RemnStr'=

	length('Str)>0			*length*('Str) =0
	'p2≤0	1 ≤ 'p2 ≤ *length*('Str)-'p1+1	*length*('Str)-'p1+1 < 'p2	
'p1<1	null	*substring*(Str', 1, min(p2', *length*(Str')))	*substring*(Str', 1, min(p2', *length*(Str')))	null
1≤ 'p1≤ *length*('Str)	null	*substring*(Str', p1', p2')	*substring*(Str', p1', *length*(Str')-p1'+1)	
length('Str) < 'p1	null	null	null	

LEXICON

Auxiliary functions

length: string → integer
length(S) $\overset{\text{df}}{=}$ The length of the sequence (string) S. The length may be zero.

substring: string × integer × integer → string
substring(S, $a1$, $a2$) $\overset{\text{df}}{=}$ The substring of the string S beginning in position $a1$ and $a2$ elements long. The null string is allowed. The values of S, $a1$ and $a2$ must satisfy the following condition:

$$a1 \in \mathbb{Z} \wedge a2 \in \mathbb{Z} \wedge 1 \leq a1 \wedge 0 \leq a2 \wedge a1 + a2 - 1 \leq length(S)$$

lim: integer × integer × integer → integer
lim(a, b, c) $\overset{\text{df}}{=}$ max(a, min(b, c))

Index