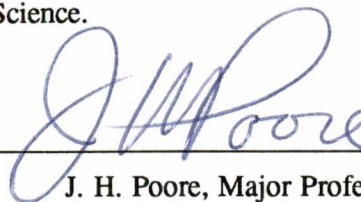


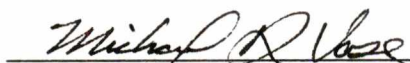
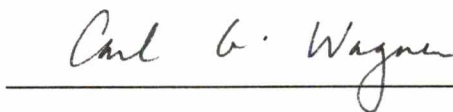
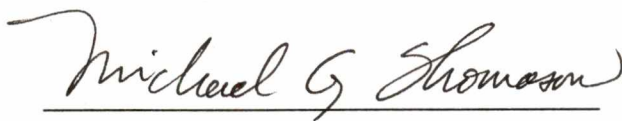
To the Graduate Council:

I am submitting herewith a dissertation written by Hailong Mao entitled "The Box-structure Development Method." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.



J. H. Poore, Major Professor

We have read this dissertation
and recommend its acceptance:



Accepted for the Council:



Associate Vice Chancellor
and Dean of The Graduate School

The Box-structure Development Method

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

Hailong Mao

December 1993

Copyright © Hailong Mao, 1993
All rights reserved

Dedication

This dissertation is dedicated to
my parents Mr. Huanqing Mao and Mrs. Lian Zhang
and
my wife Linmo and children Timmy and Jimmy

Acknowledgements

This dissertation is the result of several years of research and software development in Cleanroom Software Engineering. Many people have contributed in various ways to this research.

First, I would like to express my utmost gratitude to my advisor Professor Jesse Poore for the opportunity to work in his Software Quality Research Laboratory, for his confidence in me, and for his excellent guidance of this research. I greatly appreciate the tremendous effort and invaluable advice he has given throughout my work on the dissertation.

I am grateful to Professors Michael Thomason, Michael Vose, and Carl Wagner for their service on my committee and for their guidance, criticism, and encouragement. I thank Professor Bruce MacLennan for his careful review and valuable comments on the draft.

I join a legion of persons who owe a debt of gratitude to Professor Harlan Mills for his seminal contributions to software engineering. Professor Mills' original work on the Box Structure Method was the starting point of my research.

I thank Dr. Carmen Trammell, Dr. Jim Whittaker, and Mrs. Ethel Wittenberg for their continuous encouragement and generous support.

I thank Mr. Richard Linger, Director of IBM Cleanroom Software Technology Center, for his pioneering work in Cleanroom and his personal support and encouragement of this research.

I thank all my fellow graduate students in the SQRL, especially Randy Baker, Leon Binder, Mitchell Duitz, Daniel Fetzer, Dave Fuhrer, Scott Hopkins, Stacy Prowell, Charles Senn, and Gwen Walton, for their knowledge and friendship. A special thanks goes to Dave Primeaux.

I am grateful to IBM Westlake and Dr. Nancy Botten for funding our research program.

Finally, I thank my wife Linmo and children Timmy and Jimmy who have given me continuous love, patience, and encouragement during my five years of study and research.

Abstract

The Box Structure Method (BSM) is a process of stepwise refinement and verification that derives a system design from a specification. The final product of the BSM is a usage hierarchy of data abstractions in which each abstraction is defined in three distinct forms: black box, state box, and clear box [1, 2, 3]. This dissertation recasts and extends BSM into a formal method of syntax, semantics, principles and methods called the Box structure Development Method (BDM). The goals of BDM are to enhance the rigor of software specification, design and verification in box structures and to make the process more tractable, predictable and repeatable.

BDM provides formal definitions of well-formed types, black boxes, state boxes, clear boxes, and systems. Mills' original mathematical model for box structures is extended such that (1) the domain and range of a box are defined as the Cartesian products of domains of data types, (2) a well-formed domain predicate is added to the black box and state box to allow definition of partial functions on the "virtual domain," and (3) a well-formed mapping predicate is defined as a characteristic function for the black box and state box functions.

Equivalence relations among stimulus histories, a theory of canonical stimulus histories, the relationship between canonical histories and states, and other mathematical properties holding among the black box, state box and clear box have been defined and studied. As a result, theorems have been established to provide insights and procedures for stepwise box structure expansion in routine software engineering practice.

BDM contains a structured specification method for the specification of complex systems. The method provides for a stimulus history-based description, a canonical history-based description, domain partitioning and a specification hierarchy of black boxes.

Theorems are provided to support systematic verification of completeness and consistency for such structured specifications.

The four fundamental principles of the Box Structure Method, i.e., referential transparency, transaction closure, state migration and common services are treated formally. Transaction closure is subsumed in completeness and consistency. Referential transparency is treated as the substitutability of equals for equals. State migration is tied to a discipline of user interfaces and data accessibility. Finally, the principle of common services is formalized for both function and data abstractions.

Table of Contents

1	Introduction	1
1.1	Formal Methods in Software Engineering	1
1.2	Box Structure Method	4
1.3	Overview of BDM	8
2	The BDM Formal Language	13
2.1	Formal Symbols	14
2.2	Well Formed Types	18
2.3	Terms	23
2.4	Well Formed Formulas	24
2.5	Well Formed Black Boxes	28
2.6	Well Formed State Boxes	29
2.7	Well Formed Clear Boxes	30
2.8	Well Formed Systems	30
2.9	Deductive Apparatus	31
3	BDM Black Boxes	32
3.1	Black Box Semantics	33
3.1.1	The Application Interface	34
3.1.2	The Domain	35
3.1.3	The Range	36
3.1.4	The Mapping	37
3.1.5	Initial and Final States of Variables	37
3.2	Stimulus History Mathematics	38
3.2.1	The Representation of Stimulus Histories	38
3.2.2	Primitive Functions on Stimulus Histories	39
3.2.3	Equivalent Stimulus Histories	41
3.2.4	Canonical Stimulus Histories	43
3.3	Black Box Specification	47
3.3.1	Specification Hierarchy	49
3.3.2	Domain Partitioning	51
3.3.3	Normal Black Boxes	53
3.3.4	SH-based Specification	61
3.3.5	CH-based Specification	66
3.4	Black Box Verification	70
3.4.1	Completeness and Consistency	70
3.4.2	Specification Verification Theorems	71

4	BDM State Boxes	77
4.1	State Box Semantics	79
4.2	State Encapsulation	80
4.3	State Migration	87
4.4	State Box Design	91
4.4.1	Domain, Range and Interface	91
4.4.2	State Space	92
4.4.3	Internal Black Box	93
4.5	State Box Verification	96
5	BDM Clear Boxes	101
5.1	Clear Box Semantics	101
5.2	Usage Hierarchy	103
5.3	Prime Control Structures	104
5.3.1	Sequence Clear Box	105
5.3.2	Alternation Clear Box	105
5.3.3	Iteration Clear Box	106
5.3.4	Concurrent Clear Box	107
5.4	Clear Box Design	107
5.4.1	Concrete State Space	107
5.4.2	Procedure Definition	108
5.5	BSM Design Principles	109
5.5.1	Referential Transparency	109
5.5.2	State Migration	111
5.5.3	Common Services	112
5.6	Clear Box Verification	114
5.6.1	Function Abstraction of Clear Boxes	115
5.6.2	Function Verification of Clear Boxes	119
6	Conclusion	121
6.1	Synopsis	121
6.2	Future Work	123
	Bibliography	125
	Appendixes	130
A	BDM Examples	131
B	BDM Reserved Words	149
	Vita	150

Chapter 1

Introduction

1.1 Formal Methods in Software Engineering

A *formal language* is comprised of a set of symbols and a formation syntax, together with an interpretation. A *formal system* consists of a formal language and a deductive apparatus of axioms and inference rules [4]. When formal systems are applied to practical ends, artifacts of a system or a process in the real world are represented and manipulated as abstract strings of symbols. Structures, behavior and new facts are derived by analysis and reasoning within the formal system. *Formal methods* are mathematically-based techniques to assist in the process of system development. A formal method has a mathematical basis, such as a formal language or a formal system, and a set of principles and procedures. Formal methods have long been the mathematical foundations of engineering disciplines.

Software engineering is a relatively new discipline for the application of science and mathematics to software specification, design, verification, certification, and management. Formal methods were originally developed and practiced only by academic and research institutions. Today, there is growing interest in formal methods for real problems on an industrial scale [4].

Formal methods provide frameworks within which software engineers can specify, design and verify software systems in a mathematical and systematic, rather than ad hoc, manner [5]. When a formal method is used in specification, the formal specification is precise and unambiguous. Such a formal specification is subject to mathematical reasoning about its

completeness and consistency so that errors can be detected in the early specification phase. The formal specification also provides a basis for verifying the implementation. When a formal method is used in design, the quality of system design can be reasoned based on theorems reflecting engineering principles such as referential transparency, transaction closure, state migration, and common services. Design flaws can be revealed early in the development process that otherwise might persist in testing and be released to field use [5]. When a formal method is used in verification, the correctness of the software system can be verified through mathematical argument of the equivalence of the specification and the implementation. The advantages of using formal methods in specification, design and verification have been discussed in detail in Meyer [6] and Scharbach [7].

The use of formal methods in software development dates from the work of Dijkstra [8, 9], Hoare [10, 11], Mills [12, 13], Parnas [14, 15], and others in the 1970s. Two broad classes of formal methods are model-based methods and property-based methods. Using model-based methods, a system's behavior is defined directly by constructing a model of the system in terms of mathematical structures such as sets, relations and functions. Using property-based methods, a system's behavior is indirectly defined by stating a set of properties, usually in the form of a set of axioms, that the system must satisfy [5]. ANNA, Trace Assertion, VDM, and Z are examples of the better-known formal methods.

ANNA, Annotated Ada, is a language extension of Ada that includes facilities for formally specifying the intended behavior of Ada programs [16]. The central specification concept in ANNA is the declarative annotation which is a generalization of the constraint concept in Ada. The formal annotation that defines the function for a program module is a set of axioms in first-order logic. ANNA is a specification language rather than a development

method. It is designed to augment Ada with precise machine-processable annotations so that well-established formal specification methods can be applied to Ada programs [17].

The Trace Assertion technique for module specification was originated by Parnas [18, 19]. In a trace specification, a module's behavior is defined in terms of traces of procedure calls and return values from those calls. A trace specification consists of a set of axioms to constrain an equivalence relation on input traces, a set of canonical traces representing equivalence classes, and the output mapping for each canonical trace.

The Vienna Development Method (VDM) [20, 21] was developed at the IBM Vienna Research Laboratories during the 1970s. VDM is a model-based formal method of software development that includes formal specifications and rigorous correctness arguments in design. Strongly influenced by Dijkstra's predicate calculus based method of procedural programming [9], VDM defines the notion of proof and axiomatizes the basic theories, such as logic, functions, maps, sets, sequences, and data types [22]. It supports top-down system development through data refinement and operation decomposition [23].

The Z notation [24] is a formal specification language developed during the early 1980s by the Programming Research Group at the Oxford University Computing Laboratory. It is a model-based approach that uses set theory and first-order predicate logic as a specification language. Z is building on the success of VDM and is similar in some respects to VDM. A distinctive feature in Z is the decomposition of a specification into small pieces called schemas. The schema calculus of Z allows combination of small schemas by propositional connectives to form complex specifications [25].

The distinctive features of Z facilitate the simplification and clarification of specifications. However, they are completely irrelevant in the design phases of system implementation, for which the notation, concepts and methods of VDM are more appropriate

[26]. Thus, Hoare advises using each notation for its primary intended purpose: Z for specification and VDM for development. Others proposed various refinement calculi to be used with Z to support the whole development cycle [27].

1.2 Box Structure Method

In the search for solutions to the "software crisis," Cleanroom software engineering [28, 29] has emerged as a process capable of producing high quality software with certified reliability under statistical quality control [30-32]. Recognizing this, Gruman [33] reported that Cleanroom was proposed as a solution to software quality, reliability, and safety problems by a US congressional subcommittee's staff report [34] on federal software procurement and development. Interest in the Cleanroom software engineering process continues to grow.

Cleanroom uses the Box Structure Method (BSM) for software specification, design, and verification. BSM is a process of stepwise refinement and verification that derives a system design from a specification. The method is "based on a usage hierarchy of data abstractions in which each abstraction is defined in three distinct forms: black box (BB), state box (SB), and clear box (CB). Each of these three box structures defines identical external behavior, but with increasing internal visibility" [2].

The black box is "an external view of a system or subsystem that accepts stimuli, and for each stimulus, produces a response before accepting the next stimulus. Each response is uniquely determined by its stimulus history" [2].

The state box is "an intermediate system view that defines an internal state, namely an abstraction of the data stored from stimulus to stimulus" [2]. A state box uses the internal state "to determine the next state and response for each stimulus. The abstract state can then

be distributed in any way desired between the state and the transition behavior of the state box" [3]. A state box has an internal black box.

The clear box gives a procedural description for the internal black box of its state box. The clear box may use next-level black boxes, and will use one or more of the four basic control structures: sequence, alternation, iteration and concurrency [2].

BSM, then, provides three views of a software system: the black box view, a sequence-oriented description; the state box view, a state-oriented description; the clear box view, a procedure-oriented description. This multi-view paradigm enables specification with no implementation bias and seamless derivation of an implementation from the specification through a usage hierarchy of boxes. BSM provides a framework for an integrated formal method to cover the three major aspects of software development (specification, design and verification) and thus can potentially provide all features offered by Z and VDM combined.

Skillful use of BSM is key to the success of Cleanroom. Current BSM and its box description language (BDL) [1] are based on Parnas' usage hierarchy of modules [35] and Linger's function-theoretic design and verification method for structured programming [36]. While BSM and BDL provide a framework for software development, the method requires completion in some details and enhancement in some aspects to support a practical and repeatable process. Experience has shown that it is difficult to use BSM successfully, and there is little uniformity of process or product among those who have used BSM successfully.

The major issues to be addressed are as follows:

- (1) BSM lacks formal definition of key concepts, such as black box function, state box function, state migration, and clear box function.

- (2) There is no concept of "application interfaces" to regulate the syntax of the three views. Explicit control is needed over the external views of BB, SB, and CB, and also the relationship between the external view and the internal view of the SB and CB.
- (3) BDL does not have formal syntax and semantics for BB and SB function descriptions. The language for defining the function of a system (top-level black box) and all its lower-level black boxes and state boxes is the choice of the developer. Very often, natural language is chosen and used in a way that lacks the precision required to support rigorous specification, design, and verification. Moreover, designs must be translated into a computer-executable language.
- (4) BSM does not provide techniques for stimulus history analysis and use in black box specifications. Such techniques are needed since the essence of the black box is the external view of a system expressed solely in terms of stimulus histories.
- (5) BSM approaches specification of complex systems, whose behavior can hardly be specified in full detail on one level, in a usage hierarchy; however, little help is given for the huge task of managing the degree of abstraction or detail revealed at each level.
- (6) BSM has four basic principles: transaction closure, referential transparency, state migration and common services for quality software development. However, little assistance is given in exploiting the wisdom in these principles.

Formal definitions and procedures are needed in order to redress these difficulties and make BSM more effective, i.e., make its use more predictable and repeatable, in Cleanroom projects.

Several efforts to extend the formality and power of the current BSM have been made. Fetzer and Poore integrated BSM with Z notation [37]. They proposed a two-stage specification which used informal language for the black box specification and then used the Z

notation for state box specification. Z provides a rigorous, formal language for the inner syntax of state boxes. However, since many boxes in a system don't need a state box, the two-stage specification strategy leaves these boxes without a formal specification.

Deck [38] introduced a taxonomy of black box semantics based on properties of the system to be specified. Three classes of semantics are defined for systems of increasing complexity in their interactions with other systems in the execution environment. Addressing the relationship between the application system and the execution environment is an important step, but descriptive syntax and semantics remain to be defined.

Several design languages for BSM, such as CDL [39], Design C [40], and Ada Design Language [41], have been proposed. In each case, existing BSM artifacts are cast to meaningful form in the target programming language. The unification of BSM outer and inner syntax under C or Ada eliminates the translation from BSM design to the target programming language. However, none of these efforts has provided solutions for the above problems.

Poore has proposed a concept called Pragmatic Box Structures (PBS). The application of BSM leaves a large gap between an abstract system design and its implementation as code executing in a specific programming language and operating system environment. This concept is to recast BSM to take implementation environments into account. The concept is currently under study.

Although there is much research on BSM, no formal and comprehensive expression of BSM exists. A formal method for box structures would permit rigorous treatment of key concepts in box structure software development, and unify the currently diverging practice and extension of BSM.

1.3 Overview of BDM

This dissertation presents the foundation for a formal Box structure Development Method (BDM) which is an integrated system of syntax, semantics, principles and methods for rigorous software specification, design and verification in box structures. The thrust of this research is to propose formal definitions of key concepts and artifacts of BSM, to reveal new insights about BSM, to provide theorems and methods applicable to the daily work of BSM practitioners, and thereby to improve both software engineering theory and practice.

BDM is a function-theoretic method for software development using set theory and the first-order predicate logic. BDM was influenced by formal methods such as VDM, Z, and the Trace Assertion Method. BDM is based on the original BSM by Mills [1] and has adapted fundamental ideas from Linger [36]. However, BDM has its own originality and distinction.

BDM provides formal syntax and the semantics of well-formed formulas, types, black boxes, state boxes, clear boxes, and software systems. Key concepts of box structures have been formally defined, such as partial black box functions, the domain and range, null responses, and the mapping. Mills' original mathematical model for box structures is extended such that:

- (1) the domain and range of a box are defined as subsets of Cartesian products of data types,
- (2) a well-formed domain predicate is introduced for the black box and state box to allow definition of partial functions,
- (3) a well-formed mapping predicate is defined as a characteristic function for the black box and state box function, and
- (4) the concept of application interfaces is introduced formally.

The relationships between the function application interface and the function definition of boxes is addressed so that well-formed BB, SB and CB correctly model the real-world phenomenon of software usage and behavior. Because of the unrestricted form of BB and SB definitions in BSM, the state machine behavior of (Stimulus, Old state) $\rightarrow$ (Response, New state) was often considered as the state box function, which creates a syntax discrepancy of box interfaces in the usage hierarchy. In BDM, BB, SB and CB are defined to have syntactically identical external function views and they can be substituted one for another in specification and design. The external function views of state box and clear box are then represented by their internal well-formed black boxes that carry out the state transition and the response generation. Thus, the dilemma of interface in usage hierarchy has been resolved so that black boxes can be used in higher-level clear boxes, and later they can be replaced by their clear boxes. Such separation and substitution between black boxes and their clear boxes are very important to team development and stepwise derivation of a system implementation from the specification. BDM provides a uniform framework for software specification, design and verification in box structures; a rapprochement between mathematical concept and expression of software design.

Since the essence of the black box is its external view of a system expressed solely in terms of stimulus histories, BDM provides a list notation and a utility set of primitive list functions for representation and analysis of stimulus histories (SH) and their subhistories. Equivalence relations among stimulus histories and the concept of canonical stimulus histories (CH) are introduced. Mathematical properties of relationships among stimulus histories and their canonical histories are studied to provide insights and alternatives for black box specifications.

A structured specification method is proposed so that a complex system can be specified in a top-down, stepwise, function presentation manner. The method provides an SH-based description, a CH-based description, a domain partitioning and a specification hierarchy of black boxes in one of two forms of standardized black boxes. SH-based descriptions define black box mappings directly in terms of stimulus histories and their subhistories using BDM list notation and primitive list functions. CH-based descriptions are a specification approach in which a canonical form for the black box is first defined, and then the black box mapping is defined in terms of canonical stimulus histories. Domain partitioning allows the black box function to be presented as a composition of a set of subfunctions corresponding to a response-based partition on the black box domain. Total case predicates and partial case predicates are introduced to facilitate domain partitioning and the mapping definition for piece-wise constructed functions. A specification hierarchy enables a usage hierarchy of black boxes where higher-level black boxes describe the general structure of the system and lower-level black boxes describe more detailed mappings. A specification hierarchy supports synthesis of systems by extensive reuse of existing boxes.

The concepts of completeness and consistency are defined for specifications. Theorems are provided to support systematic verification of completeness and consistency for structured software specifications. The central ideas behind the verification theorems are:

- (1) to prove completeness and consistency by verifying a finite number of subfunctions defined by domain partitioning, and
- (2) to prove these properties for a large and complex BB specification by proving the properties for a set of smaller and less complex (lower level) black boxes.

State encapsulation from stimulus histories is explored. The relationship between canonical stimulus histories and states is analyzed. Theorems about state encapsulation

provide theoretical guidelines for determining whether states should be images of a one-to-one mapping from canonical stimulus histories or an optimal aggregation of canonical stimulus histories. Various mathematical properties and relationships among BB, SB and CB, such as state migration, total state encapsulation, partial state encapsulation and the conservation of state space during state migration, are studied. Theoretical insights and actual procedures are provided for state box design. A new approach to state box verification has been proposed wherein the state box is verified directly against its black box instead of the BSM derive-then-compare approach. The new approach avoids the difficulty of reverse mapping from state to stimulus histories and also eliminates the extra derived black box for every state box.

BDM formally defines and incorporates the four fundamental principles of the Box Structure Method, i.e., referential transparency, transaction closure, state migration and common services, so that box structure designs reflect these principles.

BDM's clear box inner syntax uses C++ syntax [42]. This facilitates BDM support of object-oriented phenomena such as class definition and inheritance. The files containing a box structure design are C++ source files as well as BDM design documents. There is no separate translation from BDM design into C++ code. The unification of outer and inner syntax makes possible a seamless transition from specification to design to code.

The remainder of this dissertation is organized as follows. Chapter 2 describes the BDM formal language which contains formal symbols, terms, well-formed types, well-formed formulas, well-formed black boxes, well-formed state boxes, well-formed clear boxes, and well-formed systems. Chapter 3 presents black boxes. Black box semantics, stimulus histories and canonical forms, the structured specification method, and the black box verification theorems are presented. Chapter 4 presents state boxes. State box semantics, state encapsulation, state migration, state box design and verification are presented. Chapter 5

presents clear boxes. Clear box semantics, prime control structures, function decomposition and clear box verification are presented. Fundamental design principles of referential transparency, state migration, and common services are formally stated. Chapter 6 gives the summary role of BDM in computer science and software engineering, summarizes the foundation that has been established and suggests future research and development issues in formal methods of software development using box structures.

Appendix A contains four examples, Stack, BirthdayBook, MinManAvg12 and SumOfSquareSinceReset. Appendix B is the list of BDM reserved words.

The foundation of BDM has been developed while designing ten increments of a software system, approximately 25,000 lines of C in total. Various interpretations of BSM were used, with each increment involving experimentation to solve problems that arose in applying BSM. BDM has evolved from the insights gained throughout this work.

Chapter 2

The BDM Formal Language

BDM uses the first-order predicate logic and set theory [43, 44] for function specification. This chapter presents the syntax and semantics of the part of the predicate logic that is essential for the BDM version of software development in box structures. The syntax, semantics and calculus of BDM basically follow Loeckx and Sieber [43]. A few formal expressions beyond standard mathematics are added, such as case terms, partial case predicates, and total case predicates, to facilitate box specification, design and verification. This chapter also describes two kinds of well-formed types, abstract types and concrete types. An important abstract type, list, with a set of operations is defined.

Backus-Naur Form (BNF) [45], is used to define the syntax of the BDM language. A few adjustments were made to BNF to make the syntax definition more readable. Nonterminal symbols are delimited by angular brackets $\langle$ and $\rangle$, while terminal symbols are not. The symbol $:=$, read as "may be composed of", is used for each production rule. Symbol $|$, read as "or", is used to combine multiple production rules for the same nonterminal symbol. Slanted braces $\{ \}$ are used to indicate repetition, e.g., $\{ x \}$ denotes zero or more occurrences of the sequence of symbols x . Slanted square brackets $[]$ are used to enclose optional symbols.

BDM's clear box inner syntax uses C++ syntax, and some lexical and sentential syntax of BDM is defined by C++ syntax. Any non-terminals with "C-" prefix are C++ language parts and its definition can be found in Ellis and Stroustrup [42]

2.1 Formal Symbols

$\langle \text{formal-symbol} \rangle := \langle \text{reserved-identifier} \rangle \mid \langle \text{identifier} \rangle \mid \langle \text{variable-symbol} \rangle \mid \langle \text{function-symbol} \rangle \mid \langle \text{predicate-symbol} \rangle \mid \langle \text{logical-symbol} \rangle \mid \langle \text{quantifier} \rangle \mid \langle \text{punctuation-symbol} \rangle$

$\langle \text{reserved-identifier} \rangle := \langle \text{BDM-reserved-identifier} \rangle \mid \langle \text{C-reserved-identifier} \rangle$

$\langle \text{BDM-reserved-identifier} \rangle := \text{See Appendix B}$

$\langle \text{identifier} \rangle := \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{digit} \rangle$

$\langle \text{letter} \rangle := _ \mid a \mid b \mid \dots \mid z \mid A \mid B \mid \dots \mid Z$

$\langle \text{digit} \rangle := 0 \mid 1 \mid \dots \mid 9$

$\langle \text{variable-symbol} \rangle := \langle \text{identifier} \rangle \mid \langle \text{identifier} \rangle^* \mid @ \langle \text{identifier} \rangle \mid \$$

$\langle \text{function-symbol} \rangle := \langle \text{constant} \rangle \mid \langle \text{n-ary-function} \rangle$

$\langle \text{constant} \rangle := * \mid \phi \mid \varepsilon \mid \omega \mid \langle \text{identifier} \rangle \mid \langle \text{integer-constant} \rangle \mid \langle \text{real-constant} \rangle \mid \langle \text{char-constant} \rangle \mid \langle \text{char-string-constant} \rangle \mid \langle \text{C-literal} \rangle$

$\langle \text{integer-constant} \rangle := [+]\langle \text{digits} \rangle \mid -\langle \text{digits} \rangle$

$\langle \text{digits} \rangle := \langle \text{digit} \rangle \mid \langle \text{digits} \rangle \langle \text{digit} \rangle$

$\langle \text{real-constant} \rangle := \langle \text{integer-constant} \rangle . \langle \text{digits} \rangle$

$\langle \text{char-constant} \rangle := \text{'}\langle \text{ascii-character} \rangle \text{'}$

$\langle \text{ascii-character} \rangle := \backslash 000 \mid \backslash 001 \mid \dots \mid \backslash 177$

$\langle \text{char-string-constant} \rangle := \text{"}\langle \text{char-string} \rangle \text{"}$

$\langle \text{char-string} \rangle := \langle \text{ascii-character} \rangle \mid \langle \text{char-string} \rangle \langle \text{ascii-character} \rangle$

$\langle \text{n-ary-function} \rangle := \langle \text{identifier-function} \rangle \mid \langle \text{operator-function} \rangle$

$\langle \text{identifier-function} \rangle := \langle \text{identifier} \rangle$

$\langle \text{operator-function} \rangle := \pi \mid \text{dom} \mid \text{ran} \mid \Delta \mid \langle \text{arithmetic-operator} \rangle \mid \langle \text{set-operator} \rangle \mid \langle \text{list-operator} \rangle \mid \langle \text{math-symbol} \rangle \mid \langle \text{C-nonlogical-operator} \rangle$

$\langle \text{arithmetic-operator} \rangle := + \mid - \mid * \mid / \mid \%$

$\langle \text{set-operator} \rangle := \# \mid \cap \mid \cup \mid - \mid \times \mid \emptyset \mid ^* \mid ^+$

$\langle \text{list-operator} \rangle := \# | + | - | \bullet | / | ^ | ^+$
 $\langle \text{predicate-symbol} \rangle := \text{TRUE} | \text{FALSE} | \langle \text{propositional-constant} \rangle | \langle \text{n-ary-predicate} \rangle$
 $\langle \text{propositional-constant} \rangle := \langle \text{identifier} \rangle$
 $\langle \text{n-ary-predicate} \rangle := \langle \text{identifier-predicate} \rangle | \langle \text{operator-predicate} \rangle$
 $\langle \text{identifier-predicate} \rangle := \langle \text{identifier} \rangle$
 $\langle \text{operator-predicate} \rangle := = | \neq | < | \leq | > | \geq | \in | \subset | \subseteq | \sqsubset |$
 $\langle \text{logical-symbol} \rangle := \neg | \wedge | \vee | \Rightarrow | \Leftrightarrow | \rightarrow | \dashv | \parallel | \langle \text{C-logical-operator} \rangle$
 $\langle \text{quantifier} \rangle := \forall | \exists | \exists_1$
 $\langle \text{punctuation-symbol} \rangle := , | . | \{ | \} | [|] | < | > | (|) | // | .. | | | :: = | ; | \langle \text{C-punctuator} \rangle$

Semantics

Identifiers are used as names for variables, data types and boxes. Identifiers are case sensitive. BDM keywords are predefined reserved identifiers that have special meanings, as listed in Appendix B.

A variable name with the decorator $^$ represents an initial state of the variable, while a variable name without a decorator $^$ represents its final state. $@$ is used as a prefix for a variable symbol to represent an abstract type imposed on the variable. $\$$ is a reserved variable name to represent the index of the last element in a list.

A function f from set A to set B , denoted $f: A \rightarrow B$, is a binary relation on $A \times B$ such that for every $a \in A$, there is at most one $b \in B$ such that $(a, b) \in f$. A is called the *domain* of the function f , and B is called the *range* of f . If $(a, b) \in f$, then we write $f(a)=b$, where b is called the *value* of the function for the *argument* a ; or b is called the *output* of function f for the *input* a ; or b is called the *image* of a under function f .

Function $f:A \rightarrow B$ is a *total function* from A to B, if for all $a \in A$, there is one and only one $b \in B$ such that $f(a)=b$. The total function f is denoted $f:A \mapsto B$.

Function $f:A \rightarrow B$ is a *partial function* from A to B, if for $A' \subseteq A$, $f:A' \mapsto B$ is a total function, but for all $a \in A-A'$, there is no $b \in B$ such that $f(a)=b$, i.e., the value of $f(a)$ is undefined [46]. In such a case, A is called the *virtual domain* of the function f , and A' is called the (*actual*) *domain* of f .

Every function has a definition and can be used many times. For function $f:A \rightarrow B$ and $a \in A$, the notation Δf represents the function definition of f . Whereas $f(a)$ represents an application of the function.

Each function has an arity, indicating number of arguments it can take. A function $f:A \rightarrow B$ is a unary function, a function of one variable. A function $f:A_1 \times A_2 \times \dots \times A_n \rightarrow B$ is a n-ary function, a function of n variables. A_i , for $1 \leq i \leq n$, is called a component set of the domain of f .

When a variable x is used as both the input variable and the output variable of a function g , the value of x may change after a mapping of g . The value of variable x before the function evaluation is called its *initial state*, denoted x° , while the value of variable x after the function evaluation is called its *final state*, denoted x , e.g., $x=g(x^\circ)$.

0-ary function symbols are constants. All C++ literals are constants and their semantics are defined in C++. An identifier as a constant is a user defined symbolic constant.

"*" is another special constant symbol which represents any legal value of a type.

" ϕ " is an empty set or empty list.

" ϵ " is read as "null" and is an overloaded constant in that the value of ϵ varies with different types.

" ω " is read as "undefined" and is an overloaded constant in that the value of ω varies with different types.

An identifier as a function symbol can be a C++ library function, a BDM primitive function or a user defined function.

An operator function can be a (prefix or suffix) unary function symbol, an (infix) binary function symbol, or an n-ary prefix function symbol.

The semantics of arithmetical operators and set operators are the classical ones.

The semantics of list operators will be defined below.

π is a projection operator. For a tuple $a=(a_1, a_2, \dots, a_n)$, $\pi_i(a) = a_i$. For a Cartesian product $A \subseteq A_1 \times A_2 \times \dots \times A_n$, $\pi_i(A)=A_i$.

dom is the domain operator such that for a function f , $dom(f)=\pi_1(f)$ is the domain of f .

ran is the range operator such that for a function f , $\pi_2(f) \subseteq ran(f)$.

Each predicate symbol has an arity, indicating the number of arguments it can take.

An identifier predicate is a predicate function that takes arguments and returns a TRUE or FALSE. An operator predicate can be an (infix) binary predicate symbol, or an n-ary (prefix) predicate symbol.

The semantics of relational operators $=$, $\neq$, $<$, $\leq$, $>$ and $\geq$ are the classical ones with component-wise extension to vectors, e.g. $(x_1, x_2, \dots, x_n) = (e_1, e_2, \dots, e_n)$ for $x_1=e_1 \wedge x_2=e_2 \wedge \dots \wedge x_n=e_n$.

The semantics of set relation operators $\in$, $\subset$ and $\subseteq$ are defined in [43]. The semantics of list relation operator $\sqsubset$ will be defined below.

The semantics of logical operators $\forall$, $\exists$, $\exists_1$, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ are the classical ones [43]. The semantics of logical symbols $\neg$ and $\parallel$ will be described below for case predicates.

Punctuation symbols have syntactic and semantic meaning when used in a BDM sentential structure to separate various syntactic elements, but do not, of themselves, specify an operation that yields a value.

, is the peer separator for objects at the same level.

. is the selection punctuator used in between a higher level variable name and a lower level variable name to select a item in a hierarchical structure, e.g., usa.tn.knoxville.

{ } is used as boundary of sets, structures, boxes or blocks of statements.

< > is used as boundary of lists.

[] and () are used in terms and formulas indicating priorities.

// is a line comment punctuator.

.. is the omission punctuator used for abbreviation of series.

| is the constraint punctuator, read as "where", used to separate a constraint condition from a term.

::= is the definition punctuator, read "is defined as", to link a symbol to its definition.

2.2 Well Formed Types (WFT)

<wft-definition> := <cdt-definition> | <adt-definition>

<cdt-definition> := <C-type-definition>

<adt-definition> := ADT <adt> <type-content>

<adt> := <identifier>

<type-content> := <set> | { / def<symbol-definition-list>for / <set>,<operations> }

<set> := <set-constant> | <set-operator><set> | <set><set-operator><set> |
{<term>|<wff>} | (<set>) | {<adt> <identifier>{,<adt> <identifier>}} |
<adt>

$\langle \text{set-constant} \rangle := \phi \mid \text{NAT} \mid \text{INT} \mid \text{REAL} \mid \text{LET} \mid \text{NUM} \mid \text{CHAR} \mid$
 $\text{BOOL} \mid \{ \langle \text{constant} \rangle, \langle \text{constant} \rangle \}$

$\langle \text{term} \rangle :=$ See section for TERMS

$\langle \text{wff} \rangle :=$ See section for WFF

$\langle \text{symbol-definition-list} \rangle :=$ See section for WFF

$\langle \text{operations} \rangle := \{ \langle \text{identifier-function} \rangle \} \mid \{ \langle \text{operator-function} \rangle \} \mid \langle \text{operations} \rangle$
 $\cup \langle \text{operations} \rangle$

ADT ϕ { }

ADT NAT { {0, 1, 2, ..}, {+, -, $\times$, /, %} }

ADT NAT₁ { {1, 2, ..}, {+, -, $\times$, /, %} }

ADT INT { {.., -1, 0, 1, ..}, {+, -, $\times$, /, %} }

ADT REAL { { $x.y \mid x \in \text{INT} \wedge y \in \text{NAT}$ }, {+, -, $\times$, /} }

ADT LET {a, b, ..., z, A, B, ..., Z}

ADT NUM {0, 1, ..., 9}

ADT CHAR { '\000', '\001', ..., '\177' }

ADT BOOL {TRUE, FALSE}

$\langle \text{wft} \rangle := \langle \text{abstract-type} \rangle \mid \langle \text{concrete-type} \rangle$

$\langle \text{abstract-type} \rangle := \langle \text{adt} \rangle \mid \langle \text{set} \rangle$

$\langle \text{concrete-type} \rangle := \langle \text{cdt} \rangle$

$\langle \text{cdt} \rangle := \langle \text{C-fundamental-type} \rangle \mid \langle \text{C-derived-type} \rangle$

$\langle \text{wftd} \rangle := \langle \text{cdt} \rangle \mid \langle \text{adt} \rangle$

$\langle \text{declaration} \rangle := \langle \text{abstract-declaration} \rangle \mid \langle \text{concrete-declaration} \rangle$

$\langle \text{abstract-declaration} \rangle := \langle \text{abstract-type} \rangle \langle \text{identifier} \rangle \{, \langle \text{identifier} \rangle \}; \mid \langle \text{identifier} \rangle \{,$
 $\langle \text{identifier} \rangle \} \in \langle \text{abstract-type} \rangle;$

$\langle \text{concrete-declaration} \rangle := \langle \text{C-declaration} \rangle$

Semantics

In BDM there are two categories of types, concrete types and abstract types. Concrete types are used in C++ programs. (Such programs are called "clear boxes" in BDM.) There are two kinds of concrete types, fundamental and derived [42]. The fundamental types are those built into C++, such as int, short, long, float, double and char. Derived types are built from fundamental types and other derived types by using the typedef and class definition statements, which allow a type to be defined as a set of values together with a set of operations on them.

Abstract types are used only in BDM black boxes and state boxes. Abstract types include classical mathematical structures represented in classical notation, which are not available in C++. There are two kinds of abstract types, simple and compound. BDM primitive abstract types, such as NAT, INT, and REAL, and other scalar types are simple types. Compound types are Cartesian products of simple and other compound types. Three important abstract types in BDM are sets, lists, and tuples.

The concept and notation of sets can be found in [43]. BDM sets are composed of elements of a single type, simple or compound. A set can be defined explicitly by enumerating its elements, or implicitly by existing types and a predicate with some free variables, e.g., $S = \{x | x \in T \wedge P(x)\}$, which is equivalent to $\forall x[x \in S \Leftrightarrow (x \in T \wedge P(x)=\text{TRUE})]$. The elements of a set can be any well formed type.

A *tuple* is an ordered set of zero or more components. The types of various components may not be the same. A *tuple type* is defined as an ordered sequence of (component-type, component-name) pairs where each pair defines a type and a variable for a component of tuples of the type. Tuple types are compound types composed of its component types. A tuple type definition $\text{ADT } A \{A_1 a_1, A_2 a_2, \dots, A_n a_n\}$, where, A_i is a well formed type and a_i is a variable, for $1 \leq i \leq n$, defines a type $A_1 \times A_2 \times \dots \times A_n$. The n-tuple variable

$a=(a_1, a_2, \dots, a_n)$ of type A is an element of $A_1 \times A_2 \times \dots \times A_n$.

A *List* is an ordered set of zero or more elements of a given type, denoted $A = \langle a_1, a_2, \dots, a_n \rangle$. The index of a list always begins with 1. ϕ is the *empty list* $\langle \rangle$ with no elements. If list L is an ordered sequence of zero or more elements from a WFT S , then $L \in S^*$. Elements of a list can be of any well formed type.

The differences between a set and a list are that there is no order among set elements but there is an order among list elements, and each element in a set is unique but an element of a list may have multiple occurrences in the list.

Operations on lists and their semantics are defined as follows. Let A and B be lists:

$a \in A$ for a is an element of A ,

$\#A$ for the cardinality of list A , i.e., the number of elements in A ,

A_i or $A[i]$ for projection of the i -th element of A ,

$\$$ represents the maximum index of A when it is involved in a list operation with A ,

$A_\$$ for projection of the last element of A ,

$A_{i..j}$ or $A[i..j]$ for projection of the sublist $\langle a_i, a_{i+1}, \dots, a_{j-1}, a_j \rangle$ of A . If $i > j$, $A_{i..j} = \phi$,

$A=B$ for A equals B such that $\#A=\#B \wedge \forall i \in \{1..\#A\} [A_i=B_i]$,

$B \subset A$ for B is a consecutive sublist of A such that $\exists i \in \{1..\#A-\#B+1\} \forall j \in \{1..\#B\}$

$$[B_j=A_{i+j}],$$

$B \subset A$ for B is a sublist of A such that $\forall b [b \in B \rightarrow b \in A]$,

$A+B$ for concatenation of A and B such that $A+B=\langle A_1, A_2, \dots, A_\$, B_1, B_2, \dots, B_\$ \rangle$,

$A-B$ for exclusion of all elements of B from A while preserving the order such that

$$\forall b \in B [b \notin (A-B)] \wedge \#(A-B) = \#A - \#\{i | A_i \in B\} \wedge$$

$$\forall i \in \{1..\#(A-B)\} \exists j \in \{1..\#A\} [i=j-\#\{x | x \in A_{1..j} \wedge x \in B\} \wedge (A-B)_i=A_j],$$

$A/(\text{index-list})$, where $\{\text{index-list}\} \in \mathcal{P}(\{1, 2, \dots, \#A\})$, deletes elements whose indices are in the index-list from list A .

List is a very important type in BDM. The essence of BSM is the external view of a software system or component solely in terms of stimulus histories, which are represented as lists in BDM. Many other commonly used types in software systems can be derived from lists. A queue is a First-In-First-Out list. A stack is a Last-In-First-Out list. A file can be considered as a list of records. A one dimensional array is a list of objects of the same type. A two dimensional array is a list of objects which are lists themselves.

A *declaration* is a sentence to introduce one or more variables and specify their types. The concrete declaration syntax and semantics can be found in [42]. Some examples of valid BDM declarations are: Real r ; $\emptyset\text{NAT } N$; INT $x, y[100]$; name $\in \text{LETTER}^+$; BOOL FileChanged, TaskDone.

Variables declared by a declaration sentence can be used only in the *scope* of the declaration. The scope of a variable depends on where and how its declaration is stated. Various kinds of scope follow the definition of C++ [42].

@ is an redefinition prefix for a variable of concrete type to be redefined as an abstract type. When a variable v is a concrete type, $@v$ will be defined as a conceptual view of the data represented by the concrete variable v . For example, let a concrete variable fp be a file pointer. Then $@fp \subseteq \{ r \mid r \in \text{RECORD_TYPE} \}$ will define $@fp$ as a set, a conceptual view, to represent the physical file pointed to by fp .

A black box may give null responses. In order to define black box syntax and semantics later, it is necessary to explicitly define the null value for various types.

For a well formed type T , its ϵ -extension, denoted T_ϵ , is defined as $T_\epsilon = T \cup \{\epsilon_T\}$, where ϵ_T is the *null value* of T and is an element that does not belong to T .

For any partial function f , $f(a)$ is not defined if $a \notin \text{dom}(f)$. It is useful to have a special "undefined value" for such $f(a)$ in order to mechanically compute terms where $f(a)$ is involved or even to transform f to a total function. Loeckx and Sieber introduced an ω -extension for undefined values [43]. Similarly, for a well formed type T , its ω -extension, denoted T_ω , is defined as $T_\omega = T \cup \{\omega_T\}$, where ω_T is the *undefined value* of T and is an element that does not belong to T .

Note that the null value ϵ is a different concept from the undefined value ω . ϵ means "nothing" but it represents a certain behavior. ω means "undefined" and represents an uncertain behavior.

2.3 Terms

$\langle \text{term} \rangle := \langle \text{constant} \rangle \mid \langle \text{variable-symbol} \rangle \mid \langle \text{term} \rangle . \langle \text{variable-symbol} \rangle \mid \langle \text{term} \rangle \langle \text{subscript} \rangle \mid \langle \text{term} \rangle \langle \text{term-separator} \rangle \langle \text{term} \rangle \mid \langle \text{term} \rangle , \dots \mid (\langle \text{term} \rangle) \mid [\langle \text{term} \rangle] \mid \{\langle \text{term} \rangle\} \mid \langle \langle \text{term} \rangle \rangle \mid \langle \text{function-application} \rangle \mid \langle \text{conditional-term} \rangle$

$\langle \text{subscript} \rangle := [\langle \text{term} \rangle] \mid \langle \text{term} \rangle$

$\langle \text{term-separator} \rangle := , \mid \dots$

$\langle \text{function-application} \rangle := \langle \text{term} \rangle^{\langle \text{term} \rangle} \mid [\langle \text{term} \rangle]_{\langle \text{term} \rangle}^{\langle \text{term} \rangle} \mid (\langle \text{term} \rangle)_{\langle \text{term} \rangle}^{\langle \text{term} \rangle} \mid \langle \text{n-ary-function} \rangle (\langle \text{term} \rangle) \mid \langle \text{operator-function} \rangle \langle \text{term} \rangle \mid \langle \text{term} \rangle \langle \text{operator-function} \rangle \mid \langle \text{term} \rangle \langle \text{operator-function} \rangle$

$\langle \text{conditional-term} \rangle := [\langle \text{case-term} \rangle] \mid (\langle \text{term} \rangle \mid \langle \text{wff} \rangle)$

$\langle \text{case-term} \rangle := \langle \text{wff} \rangle \rightarrow \langle \text{term} \rangle \mid \text{otherwise} \rightarrow \langle \text{term} \rangle \mid \langle \text{wff} \rangle \rightarrow \langle \text{term} \rangle \parallel \langle \text{case-term} \rangle$

$\langle \text{wff} \rangle :=$ See section for WFF

Semantics

The semantics of terms in BDM follows the semantics of classical predicate logic [43,44].

$[\langle \text{term} \rangle]_{\langle \text{term-list} \rangle}^{\langle \text{term-list} \rangle}$ is a *variable substitution* term. For a term W , variables s_i , and terms t_i , for $1 \leq i \leq n$, $[W]_{s_1, s_2, \dots, s_n}^{t_1, t_2, \dots, t_n}$ is the term formed by applying the Renaming Theorem ([43] page 24) until no variable in t_i 's occurs bound in W and then replacing all free occurrences of s_i in W by t_i , for $1 \leq i \leq n$.

$(\langle \text{term-list} \rangle)_{\langle \text{term-list} \rangle}^{\langle \text{term-list} \rangle}$ is a *list element substitution* term. $(L)_{j_1, j_2, \dots, j_n}^{t_1, t_2, \dots, t_n}$, where L is a list of terms, j_i is an index of L and t_i is a term, for $1 \leq i \leq n$, is the term formed by replacing the j_i th term in L by term t_i , for $1 \leq i \leq n$.

Two kinds of conditional terms are introduced, case terms and constraint terms. A case term has the appearance

$$[C_1 \rightarrow T_1 \parallel C_2 \rightarrow T_2 \parallel \dots \parallel C_n \rightarrow T_n \parallel \text{otherwise} \rightarrow T_{n+1}]$$

where $\text{otherwise} = \neg(\bigvee_{i=1..n}(C_i))$. The value of the case term depends on the value of conditions C_i 's. At a given time, only one of the conditions will be true. If C_i , for some $i \in [1..n]$, is true, the value of the conditional term is the value of T_i . When only one condition is presented in a case term, the case term $[C \rightarrow T]$ is denoted $T|C$, called constraint term.

The semantics of a case term is defined by the following equivalence:

$$x = [C_1 \rightarrow T_1 \parallel \dots \parallel C_n \rightarrow T_n] \Leftrightarrow [(C_1 \Rightarrow (x = T_1)) \wedge \dots \wedge (C_n \Rightarrow (x = T_n))] \text{ and,}$$

$$x = [C_1 \rightarrow T_1 \parallel \dots \parallel C_n \rightarrow T_n \parallel \text{otherwise} \rightarrow T_{n+1}] \Leftrightarrow [(C_1 \Rightarrow (x = T_1)) \wedge \dots \wedge (C_n \Rightarrow (x = T_n)) \wedge (\text{otherwise} \Rightarrow (x = T_{n+1}))]$$

2.4 Well Formed Formulas (WFF)

$\langle \text{wff} \rangle := \text{TRUE} \mid \text{FALSE} \mid \langle \text{propositional-constant} \rangle \mid \langle \text{term} \rangle \langle \text{operator-predicate} \rangle \langle \text{term} \rangle \mid \langle \text{n-ary-predicate} \rangle (\langle \text{term} \rangle) \mid (\langle \text{wff} \rangle) \mid [\langle \text{wff} \rangle] \mid \neg \langle \text{wff} \rangle \mid \langle \text{wff} \rangle \wedge \langle \text{wff} \rangle \mid \langle \text{wff} \rangle \vee \langle \text{wff} \rangle \mid \langle \text{wff} \rangle \Rightarrow \langle \text{wff} \rangle \mid \langle \text{wff} \rangle \Leftrightarrow \langle \text{wff} \rangle \mid \langle \text{quantifier} \rangle \langle \text{variable-range} \rangle \langle \text{wff} \rangle \mid [\langle \text{total-case-predicate} \rangle] \mid [\langle \text{partial-case-predicate} \rangle] \mid \{ \text{def} \langle \text{symbol-definition-list} \rangle \text{for} \langle \text{wff} \rangle \} \mid \text{def} \langle \text{symbol-definition-list} \rangle \text{for} \langle \text{wff} \rangle$

$\langle \text{variable-range} \rangle := \langle \text{variable-symbol} \rangle \mid \langle \text{variable-symbol} \rangle \in \langle \text{set} \rangle \mid (\langle \text{variable-range} \rangle) \mid \langle \text{variable-range} \rangle, \langle \text{variable-range} \rangle$

$\langle \text{total-case-predicate} \rangle := \langle \text{wff} \rangle \rightarrow \langle \text{wff} \rangle \mid \text{otherwise} \rightarrow \langle \text{wff} \rangle \mid \langle \text{wff} \rangle \rightarrow \langle \text{wff} \rangle \parallel \langle \text{total-case-predicate} \rangle$

$\langle \text{partial-case-predicate} \rangle := \langle \text{wff} \rangle \dashv \langle \text{wff} \rangle \mid \text{otherwise} \dashv \langle \text{wff} \rangle \mid \langle \text{wff} \rangle \dashv \langle \text{wff} \rangle \parallel \langle \text{partial-case-predicate} \rangle$

$\langle \text{symbol-definition-list} \rangle := \langle \text{symbol-definition} \rangle \mid \langle \text{symbol-definition-list} \rangle ; \langle \text{symbol-definition} \rangle$

$\langle \text{symbol-definition} \rangle := \langle \text{wft-definition} \rangle \mid \langle \text{declaration} \rangle \mid \langle \text{identifier} \rangle = \langle \text{term} \rangle \mid \langle \text{function-signature} \rangle \mid / \langle \text{function-signature} \rangle ::= / \langle \text{function-definition} \rangle \mid / \langle \text{function-signature} \rangle ::= / \langle \text{predicate-definition} \rangle \mid \langle \text{wfbb-definition} \rangle$

$\langle \text{function-signature} \rangle := \langle \text{n-ary-function} \rangle : \langle \text{set} \rangle \rightarrow \langle \text{set} \rangle \mid \langle \text{n-ary-predicate} \rangle : \langle \text{set} \rangle \rightarrow \langle \text{set} \rangle$

$\langle \text{function-definition} \rangle := \langle \text{n-ary-function} \rangle (\langle \text{argument-list} \rangle) = \langle \text{term} \rangle \mid \langle \text{operator-function} \rangle \langle \text{identifier} \rangle = \langle \text{term} \rangle \mid \langle \text{identifier} \rangle \langle \text{operator-function} \rangle \langle \text{identifier} \rangle = \langle \text{term} \rangle \mid \langle \text{identifier} \rangle \langle \text{operator-function} \rangle = \langle \text{term} \rangle$

$\langle \text{argument-list} \rangle := / \text{ wftd } / \langle \text{identifier} \rangle \mid \langle \text{argument-list} \rangle, / \text{ wftd } / \langle \text{identifier} \rangle$

$\langle \text{predicate-definition} \rangle := \langle \text{n-ary-predicate} \rangle (\langle \text{argument-list} \rangle) = \langle \text{wff} \rangle \mid \langle \text{identifier} \rangle \langle \text{operator-predicate} \rangle \langle \text{identifier} \rangle = \langle \text{wff} \rangle$

$\langle \text{wfbb-definition} \rangle :=$ See section for WFBB

Semantics

The semantics of quantified formulas $\forall x[w(x)]$ and $\exists x[w(x)]$ are formally defined in [43]. Let x be a variable, t_1 and t_2 be terms, S be a set or a type, and w be a formula containing variable x . The semantics of additional quantified formulas used in BDM are defined in terms of the semantics of $\forall x[w(x)]$ and $\exists x[w(x)]$ as follows:

$$\begin{array}{lll} \forall (x \in S)[w(x)] & \Leftrightarrow & \forall x[x \in S \Rightarrow w(x)] \\ \exists (x \in S)[w(x)] & \Leftrightarrow & \exists x[x \in S \wedge w(x)] \\ \exists_1 x[w(x)] & \Leftrightarrow & \exists x[w(x) \wedge \forall y[w(y) \Rightarrow y=x]] \\ \exists_1 (x \in S)[w(x)] & \Leftrightarrow & \exists_1 x[x \in S \wedge w(x)] \end{array}$$

In the total case predicate $[C_1 \rightarrow A_1 \parallel \dots \parallel C_n \rightarrow A_n]$, C_i 's are called condition formulas, and A_i 's are called action formulas. When a total case predicate is used in a function definition, the condition formulas C_i 's partition the domain into n mutually disjoint subdomains, and each action formula A_i defines a subfunction for a subdomain. The semantics of the total case predicates are defined by the following equivalences:

$$[C_1 \rightarrow A_1 \parallel \dots \parallel C_n \rightarrow A_n] \Leftrightarrow [\bigwedge_{i=1,n}(C_i \Rightarrow A_i)]$$

$$[C_1 \rightarrow A_1 \parallel \dots \parallel C_n \rightarrow A_n \parallel \text{otherwise} \rightarrow A_{n+1}] \Leftrightarrow [(\bigwedge_{i=1,n}(C_i \Rightarrow A_i)) \wedge (\neg(\bigvee_{i=1,n}(C_i)) \Rightarrow A_{n+1})]$$

The semantics of the case predicate indicate that implication subpredicates are commutative with respect to $\parallel$. Since the order of these subpredicates is not constrained, the commutative property provides a non-sequential mechanism to specify multi-branch transactions.

Similarly, in a partial case predicate $[C_1 \rightarrow A_1 \parallel \dots \parallel C_n \rightarrow A_n]$, C_i 's are called condition formulas, and A_i 's are called action formulas. It is called "partial" because only a subset of the output variables are defined in the action formulas.

In order to facilitate the description of the semantics of partial case predicates, consider a function r to yield a list of variables whose final states are defined in a formula as follows.

Let W be the set of all well formed formulas and V be the set of all variables.

Function r is defined as $r: W \rightarrow \mathcal{P}(V)$, where for a given WFF w , $r(w) = \{ v \mid v \in V \wedge \text{final state of } v \text{ occurs in } w \}$.

Let $P(x_1, \dots, x_m, y_1, \dots, y_n)$ be an $(m+n)$ -ary predicate formula as the characteristic function for a function $(y_1, \dots, y_n) = f(x_1, \dots, x_m)$. Let $Y = \{y_1, \dots, y_n\}$ be output variables. If p is a subformula of P and $Y \subseteq r(p)$, then Y is said to be totally defined in p , otherwise it is partially defined in p .

Now, let the partial case predicate $[C_1 \rightarrow A_1 \parallel C_2 \rightarrow A_2 \parallel \dots \parallel C_n \rightarrow A_n]$ be a subformula in the above defined characteristic function P. Let "otherwise" be $C_{n+1} = \neg(C_1 \vee \dots \vee C_n)$. The semantics of the partial case predicate are defined by the following equivalences:

$$[C_1 \rightarrow A_1 \parallel \dots \parallel C_n \rightarrow A_n] \Leftrightarrow [[\bigwedge_{i=1,n}(C_i \Rightarrow (A_i \wedge (Y-r(A_i))=(Y-r(A_i))^*))] \wedge [C_{n+1} \Rightarrow (Y=Y^*)]]$$

$$[C_1 \rightarrow A_1 \parallel \dots \parallel C_n \rightarrow A_n \parallel \text{otherwise} \rightarrow A_{n+1}] \Leftrightarrow [\bigwedge_{i=1,n+1}(C_i \Rightarrow (A_i \wedge (Y-r(A_i))=(Y-r(A_i))^*))]$$

where $(Y-r(A_i))$ is the set of output variables whose final states are not defined in A_i , and $(Y-r(A_i))^*$ are the initial states of such output variables before performing the subfunction.

Unlike the total case predicate, in each action formula A_i , of a partial case predicate, one does not have to explicitly define the final state of every output variable. Only the final states of those output variables of interests are explicitly defined. The final states of the other output variables remain unchanged by the implication of the partial case predicate.

If in a partial case predicate, each action formula A_i , does explicitly define the final state of every output variable, i.e., $\forall i \in \{1..n\} [Y \subseteq r(A_i)]$, then

$$[C_i \rightarrow A_i] \Leftrightarrow [C_i \rightarrow (A_i \wedge (Y-r(A_i))=(Y-r(A_i))^*)] \Leftrightarrow [C_i \rightarrow (A_i \wedge \phi=\phi^*)] \Leftrightarrow [C_i \rightarrow A_i].$$

Thus, $[C_1 \rightarrow A_1 \parallel \dots \parallel C_n \rightarrow A_n \parallel \text{otherwise} \rightarrow A_{n+1}] \Leftrightarrow [C_1 \rightarrow A_1 \parallel \dots \parallel C_n \rightarrow A_n \parallel \text{otherwise} \rightarrow A_{n+1}]$, i.e., in such a case, the partial case predicate is equivalent to a total case predicate.

Usually, boxes at higher levels of the usage hierarchy are complex and involve many transactions and output variables. Partial case predicates enable each transaction to only explicitly define output variables of interest and leave the rest of the output variables implicitly defined. Also in a function with error handling, partial case predicates enable the normal case to only explicitly define outputs related to the normal behavior, while the error handling case only takes care of the outputs related to error handling. Hence, the partial case predicates provide a powerful and convenient way to specify complex functions.

The <symbol-definition-list> is used to define some local types, variables, functions, predicates, or even some specification black boxes. The scope of a local definition is determined by punctuation if it does not extend to the end of the box in which it is used.

The classical interpretations of wffs are subject to ω -extension in BDM. Thus, for a partial function f of type t appearing in a wff w , if $f(d)$ is undefined, then $f(d)=\omega_t$ is the ω -extension used to evaluate w .

2.5 Well Formed Black Boxes (WFBB)

```
<wfbb-definition> := BB <return-type> <box-name>( in <SL>, out <RL> )
                    {
                      [ dom      < $\delta$ > ]
                      [ map /    < $\chi$ > ]
                    }
```

<return-type> := <adt> | <C-decl-specifiers>

<box-name> := <identifier>

<SL> := <wftd-argument-declaration-list>

<RL> := <wftd-argument-declaration-list>

<wftd-argument-declaration-list> := <wftd-argument-declaration> | <wftd-argument-declaration-list>, <wftd-argument-declaration>

<wftd-argument-declaration> := [<adt> / <identifier> | <C-argument-declaration>

< δ > := <wff>

< χ > := <wff>

Semantics

Black boxes are specifications for software systems or components that are based on the observation that a program is a mathematical function, and that it is productive to treat it

as such. Black boxes emphasize what must be done rather than how it is to be done. Chapter 3 is devoted to the semantics of the black box.

2.6 Well Formed State Boxes (WFSB)

```
<wfsb-definition> := SB <return-type> <box-name>( in <SL>, out <RL> )
                        {
                            state    <QL-q1>
                            / dom    <δ> /
                            map      <χ>
                        }
```

<return-type> := <C-decl-specifiers>

<box-name> := <identifier>

<SL> := <C-argument-declaration-list>

<RL> := <C-argument-declaration-list>

<QL-q₁> := { <wft> <identifier>=<initial-state-component>; } | { static <C-declaration-statement>; }

<initial-state-component> := <constant> | <C-literal>

<δ> := <wff>

<χ> := <wff>

Semantics

A state box is a state machine representation of a black box. Its role is to serve as a stepping stone from a functional specification to code (clear box), by facilitating the discovery of the necessary and sufficient information that must be captured from stimulus histories as state data or to serve as a state-based specification in the event the black box is intractable.

Chapter 4 is devoted to the semantics of the state box.

2.7 Well Formed Clear Boxes (WFCB)

```
<wfcb-definition> := CB <return-type> <box-name>( in <SL>, out <RL> )  
    {  
        state    <TL-t1>  
        proc     <local-declarations>  
                <procedure-body>  
    }
```

<return-type> := <C-decl-specifiers>

<box-name> := <identifier>

<SL> := <C-argument-declaration-list>

<RL> := <C-argument-declaration-list>

<TL-t₁> := { static <C-declaration-statement>; } | NONE

<local-declarations> := { <C-declaration-statement>; }

<procedure-body> := <C-statement-list>

Semantics

A clear box is a computer program which computes the function specified in the black box. The state box representation of the black box will constrain the data structures and algorithms used in the clear box. Chapter 5 is devoted to the semantics of the clear box.

2.8 Well Formed Systems

<system> := <BDM-files>

<BDM-files> := <BDM-file> | <BDM-files><BDM-file>

<BDM-file> := <BDM-components>

<BDM-components> := <BDM-component> | <BDM-components><BDM-component>

<BDM-component> := <C-control-line> | <C-comment> | <wft-definition> | <declaration> |
 <wfb-definition> | <wfsb-definition> | <wfcb-definition>

Semantics

A BDM well formed system is based on a hierarchy of box structures, where each clear box is defined by using a fixed set of control structures and using the specifications of lower level black boxes.

2.9 Deductive Apparatus

The deductive apparatus consists of a set of axioms and a set of inference rules. A deductive apparatus enables the derivation of new assertions from the specification in a way that can be proved formally. Also, the deductive apparatus enables reasoning about the specification as syntactic structures with no reference to any particular interpretation of the formal language. Loeckx and Sieber [43] have proved that all the formulas derivable from a specification based on a deductive system are also logical consequences of the specification based on any interpretation.

The deductive apparatus of BDM uses axioms and inference rules in the first-order predicate calculus, formal number theory, and axiomatic set theory [44]. A *derivation* in BDM is a finite sequence of statements, each of which is either an axiom, a theorem, an assertion contained in the original specification, an initial state of input variables, or an assertion concluded from previous statements in the derivation by some inference rules. An assertion A is *derivable* from a specification Δf , denoted as $\Delta f \vdash A$, if A is the last statement of some derivation from Δf .

Specifications in BDM are formal using a set of syntactical objects, such as well-formed formulas, black boxes, and state boxes. For a logically valid input, the logical consequence derived from the specification is the response to that input. Derivations will be used during black box, state box and clear box verification which will be discussed in Chapters 3, 4 and 5, respectively.

Chapter 3

BDM Black Boxes

This chapter presents the semantics of well formed black boxes, the stimulus history mathematics, the black box specification method, and the black box verification theorems.

Mills [3] defined the black box as a mathematical function $f: S^* \rightarrow R$, where S is the set of all possible stimuli and R is the set of all possible responses. His intention was that the specification for a software system or component should be described in such terms without regard to implementation details. Any effort to write a specification in this way will quickly reveal the need for many additional details regarding characteristics of the inputs, outputs and mappings. Furthermore, when this is done in consideration that code will be written later to implement the specification, additional issues become important even with such a high-level view of the specification.

A BDM black box is defined as a partial function from a set of stimulus histories to a set of responses. Basic components of a black box, the domain, range and mapping are defined, as well as BDM features of application interface and the null response.

Since the essence of the box structure method is its sequence-based external view of a software system, a list notation and a utility set of primitive list functions are provided for representation and analysis of stimulus histories and their subhistories. An equivalence relation on stimulus histories (SH) and the concept of canonical stimulus histories (CH) are introduced. Mathematical properties of relationships among stimulus histories and their

equivalent canonical histories are studied to provide insights and alternatives for black box specifications.

The structured specification method of BDM provides an SH-based description, a CH-based description, a domain partitioning, and a specification hierarchy of black boxes. BDM enables the specification of a complex system to be presented in a top-down and stepwise manner and supports synthesis of systems by extensive reuse of existing boxes.

The concepts of completeness and consistency are defined for BDM specifications. Theorems and algorithms are provided to support systematic verification of completeness and consistency for BDM structured black box specifications.

3.1 Black Box Semantics

Definition A well formed black box, denoted f_b , has definition denoted Δf_b , and

$\Delta f_b = (SL, RL, \delta, \chi)$, where $SL = (tS_1 S_1, tS_2 S_2, \dots, tS_m S_m)$ is a finite list of *stimulus variable* declarations, where for $1 \leq i \leq m$, S_i is a stimulus variable and tS_i is the type of S_i . $RL = (tR_1 R_1, tR_2 R_2, \dots, tR_n R_n)$ is a finite list of *response variable* declarations, where for $1 \leq i \leq n$, R_i is a response variable and tR_i is the type of R_i .

$\delta: S^+ \rightarrow \{\text{TRUE}, \text{FALSE}\}$ is the domain predicate, and

$\chi: S^+ \times R \rightarrow \{\text{TRUE}, \text{FALSE}\}$ is the mapping predicate .

There are two kinds of black boxes, *regular black box* and *specification black box*. A regular black box is implemented by a clear box as an executable program. A specification black box is used in other black boxes or state boxes only for specification purpose without being implemented as a clear box. If not explicitly stated otherwise, "black box" refers to a regular black box.

3.1.1 The Application Interface

Definition In the BDM syntax of <wfb-definition>, the part

<box-name>(in <SL>, out <RL>) is called the *application interface*.

The application interface of a black box prescribes how the box will be mentioned. Black box names should follow a user designated naming convention, including a means of distinguishing specification black boxes from regular black boxes, e.g., the "_" might be the initial symbol of names of specification functions.

If the black box returns a value when it is used in an expression, the BDM reserved word "Return" will be defined as the last (logical) response variable in RL and will be used in the mapping predicate to represent the function return value.

One of the fundamental principles of BDM is that a black box and both its state box and clear box, have exactly the same interface and the same function. Since a clear box is an executable program, concrete types must be used in its interface. Therefore, the interface variables of a regular black box must also be of concrete type.

There may be occasions when variables of concrete type in a black box interface can not be fully determined until its clear box is defined. Then, the type definition for interface variables can be deferred until the design is sufficiently developed to allow specification of data types and structures. In such cases, there are iterations of BB-SB-CB expansion.

However, when the definitions for BB, SB and CB of a box are completed, all three forms of a box should have exactly the same interface.

Since specification black boxes are defined to facilitate specification writing and used only in other black boxes and state boxes, the types of their interface variables can be either concrete or abstract.

For a black box f_b with $SL = (iS_1 S_1, iS_2 S_2, \dots, iS_m S_m)$ and

$RL = (tR_1 R_1, tR_2 R_2, \dots, tR_n R_n)$, f_b will be used in clear boxes by such syntax as:

$f(\text{in } S_1, S_2, \dots, S_m, \text{out } R_1, R_2, \dots, R_n)$, where $R_n \neq \text{"Return"}$, or $\text{Return} = f(\text{in } S_1, S_2, \dots, S_m, \text{out } R_1, R_2, \dots, R_{n-1})$, where $R_n = \text{"Return"}$ and formal parameters $S_1, S_2, \dots, S_m, R_1, R_2, \dots, R_n$ will be replaced by actual values.

When f_b is used in other black boxes or state boxes, the following two applications are equivalent:

$$f(\text{in } S_1, S_2, \dots, S_m, \text{out } R_1, R_2, \dots, R_n)$$

$$\Leftrightarrow (R_1, R_2, \dots, R_n) = f(S_1, S_2, \dots, S_m), \text{ where } R_n \neq \text{"Return"}, \text{ or}$$

$$\text{Return} = f(\text{in } S_1, S_2, \dots, S_m, \text{out } R_1, R_2, \dots, R_{n-1})$$

$$\Leftrightarrow (R_1, R_2, \dots, R_{n-1}, \text{Return}) = f(S_1, S_2, \dots, S_m), \text{ where } R_n = \text{"Return"}.$$

3.1.2 The Domain

The black box domain predicate δ is a BDM well formed formula. The domain of a black box is defined by the stimulus variable list SL and δ .

The set of all possible *stimuli* is $S = t(S_1)_\epsilon \times t(S_2)_\epsilon \times \dots \times t(S_m)_\epsilon$, where for $1 \leq i \leq m$, $t(S_i)_\epsilon = t(S_i) \cup \{\epsilon_{t(S_i)}\}$, an ϵ -extension of $t(S_i)$. A single stimulus is an m -tuple in S . Some components of a stimulus may have the null value ϵ to model the phenomenon that members of these m different stimulus types may appear in various combinations to make up a single stimulus. A member of S^* is called a *stimulus history*.

A BDM black box is a partial function from S^* , the set of all stimulus histories of length one or greater, to R . S^* is called the *virtual domain*. The domain of the black box function, denoted SH , is a subset of S^* constrained by the domain predicate δ such that $SH = \{ sh \mid sh \in S^* \wedge \delta(sh) \}$.

Thus, δ can be used to define any subset of S^+ as the domain SH of the black box. If SH is a proper subset of S^+ , then some stimulus histories do not belong to SH even though each individual stimulus of them may be valid.

Let $VS = \{ s \mid s \in S \wedge \exists h \in S^+ [\delta(h+\langle s \rangle)] \}$ be the set of all valid stimuli. The following are some specific situations of domain definitions of black box functions.

If δ has constant value TRUE, then $SH = \{ sh \mid sh \in S^+ \wedge \text{TRUE} \} = S^+$. The black box function is a total function from S^+ to R which is determined by the type definition of stimulus variables in SL.

If $SH=VS^+$, then the BDM domain is the same as in Mills' formulation.

If $SH=S^+ \times VS$, then the stimulus history is in the domain though it may contain some invalid stimuli before the current stimulus.

Deck [38] and Becker[47] have introduced some special sets in addition to Mills' original S , the set of all possible stimuli, to represent some special types of stimuli based on some specific application domains and environments, such as system interaction with operating system services and concurrent computation. Each of the models proposed takes care of some specific application domain. In BDM, S is characterized by a Cartesian product of one or more component sets $tS_1, tS_2, \dots, tS_m$, where the number of stimulus variables and their types depend on the application to be defined.

3.1.3 The Range

The response list RL defines $R = t(R_1)_\epsilon \times t(R_2)_\epsilon \times \dots \times t(R_n)_\epsilon$ as the range of the black box function, where for $1 \leq i \leq n$, $t(R_i)_\epsilon = t(R_i) \cup \{\epsilon_{t(R_i)}\}$, an ϵ -extension of $t(R_i)$.

Mills allowed the concept of a null response from a black box [2], but the technique to include this null response in range R and propagate it through the state box and clear box was not prescribed.

In BDM, the null response is precisely defined. The range of the black box is a Cartesian product of response types with ϵ -extension. Thus, for a response $r \in R$, any of the components of r may be null. If $r = \epsilon = (\epsilon_{r(R_1)}, \epsilon_{r(R_2)}, \dots, \epsilon_{r(R_n)})$, r is called a null response.

3.1.4 The Mapping

The black box mapping predicate χ is a well formed formula.

$\chi: SH \times R \rightarrow \{TRUE, FALSE\}$ is the characteristic function for black box function f_b such that $(sh, r) \in f_b \Leftrightarrow \chi(sh, r) = TRUE$. Thus, Δf_b defines a function $f_b: SH \rightarrow R$ such that $f_b = \{ (sh, r) \mid sh \in SH \wedge r \in R \wedge \chi(sh, r) \}$.

Black box f_b is a partial function on $S^+ \times R$. However, it is always possible to modify the definition to transform f_b to a total function $f'_b: S^+ \rightarrow R_\omega$ by defining:

$$R_\omega = R \cup \{\omega_R\}, \text{ and}$$

$$f'_b = f_b \cup \{ (sh, \omega_R) \mid sh \in (S^+ - SH) \}$$

where, ω_R is an undefined value for set R . For example, ω_R can be a message "output for this input is not defined". The black box mapping χ is an application-oriented and non-procedural function description.

3.1.5 Initial and Final States of Variables

A variable has many attributes, such as its type, its value, and its physical address.

When a variable appears in a black box definition, it represents its value. The state of a variable before evaluating a function is called the initial state of the variable, while the state of

a variable after evaluating a function is called the final state of the variable. The initial state of a variable is represented by the variable name followed by decorator $^{\circ}$. When decorator $^{\circ}$ is applied to a list of variables, the semantics are defined as follows:

$$(x_1, x_2, \dots, x_n)^{\circ} \equiv (x_1^{\circ}, x_2^{\circ}, \dots, x_n^{\circ})$$

$$\phi^{\circ} = \phi$$

Each external variable of a black box can be classified as a pure input, a pure output or an inout variable. Let SRL represent the set $SL \cap RL$. Then, the three categories are represented by ordered sets $SL-SRL$, $RL-SRL$ and SRL , respectively. The decorator $^{\circ}$ is mostly applied to external variables that are both input and output. We have the following axioms about initial states and final states of variables,

$$\forall v \in (SL-SRL)[v = v^{\circ}] \quad \text{pure stimuli are unchanged}$$

$$\forall v \in (RL-SRL)[v^{\circ} = \epsilon] \quad \text{pure responses have no initial states}$$

3.2 Stimulus History Mathematics

3.2.1 The Representation of Stimulus Histories

Since a stimulus history is a sequence of stimuli to a black box, it is naturally represented in the BDM list notation. List operations defined in BDM can be applied to stimulus histories and their subhistories. Let $\Delta f_b = (SL, RL, \delta, \chi)$, where $SL = (S_1, S_2, \dots, S_m)$ contains m stimulus variables. A stimulus history of f_b , denoted sh , is a list $\langle s_1, s_2, \dots, s_n \rangle$, where s_i , for $1 \leq i \leq n$, is a stimulus, a member of $\tau(S_1)_{\epsilon} \times \tau(S_2)_{\epsilon} \times \dots \times \tau(S_m)_{\epsilon}$. Some components of a stimulus may be null. A stimulus can be compressed by removing its null components as long as no ambiguities are introduced.

The last stimulus s_n of sh is called the *current stimulus*. The index of the current stimulus is equal to $\#sh$. $h = \langle s_1, s_2, \dots, s_{n-1} \rangle$ is called the *prior stimulus history*. Thus, $sh = h + \langle s_n \rangle$. The general form of a stimulus history of length n is

$$\langle (S_{11}, S_{21}, \dots, S_{m1}), (S_{12}, S_{22}, \dots, S_{m2}), \dots, (S_{1n}, S_{2n}, \dots, S_{mn}) \rangle,$$

where S_{ij} represents the value of the i -th stimulus variable in the j -th stimulus. Elements of the current stimulus are usually just represented by stimulus variable names $S_1, S_2, \dots, S_m$ without the index n .

In the specification of a particular black box, stimulus variables $S_1, S_2, \dots, S_m$ will be real names in the application. For example, a black box *Stack* might have the stimulus variable list (Command, InValue), where $\text{Command} \in \{\text{CREATE}, \text{PUSH}, \text{POP}, \text{TOP}\}$ and $\text{InValue} \in \text{INT}$. The general form of stimulus histories is

$$\langle (\text{Command}_1, \text{InValue}_1), (\text{Command}_2, \text{InValue}_2), \dots, (\text{Command}_n, \text{InValue}_n) \rangle,$$

where Command_j represents the value of Command in the j -th stimulus of the stimulus history. The current stimulus is $(\text{Command}_n, \text{InValue}_n)$, or simply (Command, InValue). With compression of null components in stimuli, some possible stimulus histories of various length are

$$\langle (\text{PUSH}, 3), (\text{PUSH}, 5), (\text{TOP}) \rangle$$

$$\langle (\text{POP}), (\text{PUSH}, 8), (\text{TOP}), (\text{PUSH}, 13) \rangle$$

$$\langle (\text{TOP}), (\text{PUSH}, 21), (\text{POP}), (\text{PUSH}, 34), (\text{POP}) \rangle$$

where Command_j in the three stimulus histories are PUSH, POP, and TOP, respectively.

3.2.2 Primitive Functions on Stimulus Histories

A set of primitive functions for list manipulations is provided to facilitate representation and analysis of stimulus histories and their subhistories. These primitive list

functions are defined as functions of a generic list. The list functions are defined in BDM syntax of well formed black boxes. ' _ ' represents an arbitrary element in a list and relational operator $\theta \in \{<, >\}$.

(1) Return the maximum element of list L

BB Max(in L, out Return)

{ [$L = \phi \rightarrow \text{Return} = \epsilon$ || otherwise $\rightarrow \text{Return} = (a \mid a \in L \wedge \forall b \in L (a \geq b))$] }

(2) Return the minimum element of list L

BB Min(in L, out Return)

{ [$L = \phi \rightarrow \text{Return} = \epsilon$ || otherwise $\rightarrow \text{Return} = (a \mid a \in L \wedge \forall b \in L (a \leq b))$] }

(3) Count the number of elements in list L that satisfies condition predicate P

BB Count(in P, L, out Return)

{ $\text{Return} = \#(a \mid a \in L \wedge [P]_a)$ }

(4) Return the index of the first element in list L that satisfies condition predicate P

BB First(in P, L, out Return)

{ $\text{Return} = \text{Min}(\{j \mid [P]_{L_j}\})$ }

(5) Return the index of the i-th element from the first in list L that satisfies predicate P

BB First(in P, L, i, out Return)

{ [$\exists k [\text{Count}(P, L_{1..k}) = i - 1 \wedge k = \text{First}(P, L_{k..s})] \rightarrow \text{Return} = k$ || otherwise $\rightarrow \text{Return} = \epsilon$] }

(6) Return the index of the last element in list L that satisfies condition predicate P

BB Last(in P, L, out Return)

{ $\text{Return} = \text{Max}(\{j \mid [P]_{L_j}\})$ }

(7) Return the index of the i-th element from the last in list L that satisfies predicate P

BB Last(in P, L, i, out Return)

{ [$\exists k [\text{Count}(P, L_{k+1..s}) = i - 1 \wedge k = \text{Last}(P, L_{1..k})] \rightarrow \text{Return} = k$ || otherwise $\rightarrow \text{Return} = \epsilon$] }

(8) Return the sublist of list L that satisfies condition predicate P

BB Sublist(in P, L, out Return)
{ Return = L - $\langle a \mid a \in L \wedge \neg[P]_a \rangle$ }

(9) Return the sum of elements in list L.

BB Sum(in L, out Return)
{ Return = $\sum_{i=1, \#L} L_i$ }

(10) Return the reverse list of list L

BB Reverse(in L, out Return)
{ #Return = #L $\wedge \forall i \in \{1.. \#L\} [Return_i = L_{\#L+1-i}]$ }

(11) Return the set of all permutations of list L

BB Perm(in L, out Return)
{ Return = { M | #M = #L $\wedge \forall a \in L [Count(_ = a, L) = Count(_ = a, M)]$ } }

(12) Return the sorted list of list L according to relational operator θ and the designated key

BB Sort(in L, θ , out Return) { Return $\in Perm(L) \wedge \forall i \in \{2.. \#L\} [Return_{i-1} \theta Return_i]$ }
BB Sort(in L, θ , Key, out Return) { Return $\in Perm(L)$
 $\wedge \forall i \in \{2.. \#L\} [Return_{i-1}.Key \theta Return_i.Key]$ }

3.2.3 Equivalent Stimulus Histories

Parnas [19] gives an important equivalence relation on input traces and defines a canonical trace concept which is the essence of his trace assertion method of module interface specification. Since there are some similarities between Parnas' traces and BDM stimulus histories, similar concepts have been explored for BDM.

Definition Let sh , and h be any non-empty stimulus histories of black box f_b . $f_b(sh)$ is the current response of f_b for sh . $f_b(sh+h)$ is called the *future behavior* of f_b for sh extended by h .

Parnas' concept of equivalence relation on traces requires that two equivalent traces must have the same current response and the same future behavior. In BDM, the equivalence relation on stimulus histories of a black box f_b is based on the future behavior, but not on the current response. This difference is key to the black box canonical theory and state box morphism theory developed here.

Behavior-based (or property-based) specification methods depend upon the specifier's ability to know and assert the equivalence of certain sequences of input with respect to the future behavior of the system. Knowing this equivalence, a generally simpler core of the specification can be established, from which the behavior of the system for arbitrary sequences of inputs may be determined. Thus, the motivation for the following series of definitions and theorems is to simplify the definition of the mapping predicate for black boxes and, thereby, to simplify the related software system specifications.

Definition (*Equivalent Stimulus Histories*) For a black box f_b with SH and S^+ , two stimulus histories $sh1$ and $sh2$ are equivalent with respect to f : $SH \rightarrow R$, denoted $sh1 \equiv sh2$, if and only if, for all $h \in S^+$ $[(sh1+h \in SH) = (sh2+h \in SH) \wedge (sh1+h \in SH \Rightarrow f(sh1+h) = f(sh2+h))]$.

Theorem The relation $\equiv$ defined above is an equivalence relation.

Proof Reflexive: For all $sh \in SH$, and for all $h \in S^+$

$$[(sh+h \in SH) = (sh+h \in SH) \wedge (sh+h \in SH \Rightarrow f(sh+h) = f(sh+h))], \text{ thus}$$

$$sh \equiv sh$$

Symmetric: For $sh1, sh2 \in SH$, if $sh1 \equiv sh2$, then for all $h \in S^+$

$$\begin{aligned}
& [(\text{sh1}+\text{h} \in \text{SH})=(\text{sh2}+\text{h} \in \text{SH}) \wedge (\text{sh1}+\text{h} \in \text{SH} \Rightarrow f(\text{sh1}+\text{h})=f(\text{sh2}+\text{h}))] \\
\Rightarrow & [(\text{sh2}+\text{h} \in \text{SH})=(\text{sh1}+\text{h} \in \text{SH}) \wedge (\text{sh2}+\text{h} \in \text{SH} \Rightarrow f(\text{sh2}+\text{h})=f(\text{sh1}+\text{h}))] \\
\Rightarrow & \text{sh2} \equiv \text{sh1}
\end{aligned}$$

Transitive: For $\text{sh1}, \text{sh2}, \text{sh3} \in \text{SH}$, if $\text{sh1} \equiv \text{sh2}$ and $\text{sh2} \equiv \text{sh3}$, then for all $\text{h} \in \text{S}^+$

$$\begin{aligned}
& [(\text{sh1}+\text{h} \in \text{SH})=(\text{sh2}+\text{h} \in \text{SH}) \wedge (\text{sh1}+\text{h} \in \text{SH} \Rightarrow f(\text{sh1}+\text{h})=f(\text{sh2}+\text{h}))] \wedge \\
& [(\text{sh2}+\text{h} \in \text{SH})=(\text{sh3}+\text{h} \in \text{SH}) \wedge (\text{sh2}+\text{h} \in \text{SH} \Rightarrow f(\text{sh2}+\text{h})=f(\text{sh3}+\text{h}))] \\
\Rightarrow & [(\text{sh1}+\text{h} \in \text{SH})=(\text{sh2}+\text{h} \in \text{SH}) \wedge (\text{sh2}+\text{h} \in \text{SH})=(\text{sh3}+\text{h} \in \text{SH}) \wedge \\
& (\text{sh1}+\text{h} \in \text{SH} \Rightarrow f(\text{sh1}+\text{h})=f(\text{sh2}+\text{h})) \wedge (\text{sh2}+\text{h} \in \text{SH} \Rightarrow f(\text{sh2}+\text{h})=f(\text{sh3}+\text{h}))] \\
\Rightarrow & [(\text{sh1}+\text{h} \in \text{SH})=(\text{sh2}+\text{h} \in \text{SH}) \wedge (\text{sh2}+\text{h} \in \text{SH})=(\text{sh3}+\text{h} \in \text{SH}) \wedge \\
& ((\text{sh1}+\text{h} \in \text{SH}) \wedge (\text{sh2}+\text{h} \in \text{SH})) \Rightarrow (f(\text{sh1}+\text{h})=f(\text{sh2}+\text{h})) \wedge (f(\text{sh2}+\text{h})=f(\text{sh3}+\text{h}))] \\
\Rightarrow & [(\text{sh1}+\text{h} \in \text{SH})=(\text{sh3}+\text{h} \in \text{SH}) \wedge (\text{sh1}+\text{h} \in \text{SH} \Rightarrow f(\text{sh1}+\text{h})=f(\text{sh3}+\text{h}))] \\
\Rightarrow & \text{sh1} \equiv \text{sh3}
\end{aligned}$$

□

The set of all possible stimulus histories can be partitioned based on this equivalence relation. As a consequence of the definition of the equivalence relation, two equivalent stimulus histories are interchangeable as prefixes of any other stimulus history. For example, the following are equivalent: $\langle (\text{PUSH}, 1), (\text{PUSH}, 2) \rangle$, $\langle (\text{PUSH}, 1), (\text{TOP}), (\text{PUSH}, 2) \rangle$, $\langle (\text{PUSH}, 1), (\text{PUSH}, 2), (\text{TOP}) \rangle$, $\langle (\text{PUSH}, 1), (\text{PUSH}, 3), (\text{POP}), (\text{PUSH}, 2) \rangle$. Furthermore, equivalence is preserved after suffixing of constant stimulus histories. However, it can be shown by examples that equivalence is not preserved by prefixing or infixing a constant stimulus history.

3.2.4 Canonical Stimulus Histories

In order to establish equivalency among stimulus histories of a black box, the effect of each stimulus type on future behavior of the black box must be analyzed and classified. Not

all stimuli in a stimulus history contribute to the determination of future responses. This is caused by various reasons. Some stimuli have no impact on the future behavior at all, such as a stimulus (TOP) to black box Stack. Some stimuli by themselves do have influence on future behavior, but their influence may be canceled by some other stimuli which appear later in the stimulus history. For example, the influence of a PUSH operation on the future behavior of a stack will be canceled if it is immediately followed by a POP operation.

Definition For a black box f_b with domain SH and valid stimuli VS, an *essential stimulus* set,

ES, is a subset of VS such that for any $sh \in SH$, there exists an $eh \in ES^*$ for which $sh \equiv eh$.

Stimuli in ES form a basis of essential information, either data information or control information, for future behavior of a black box. Semantics of any stimulus history in the domain SH can be represented by a sequence of essential stimuli. A stimulus in VS-ES can be either simply omitted or replaced by some stimuli from ES. More than one ES may exist for a given a black box. It is an analysis and invention step in specification writing to identify the essential sets of stimuli and to decide which one will best represent all semantics of the domain SH.

The set of stimulus histories of a black box can be partitioned by the equivalence relation. In each equivalence class, the individual stimulus histories still have important and distinctive qualities, e.g. length, though they all produce the same response.

Parnas [19] described "canonical traces" to represent equivalence classes and to facilitate module specifications in the trace assertion method. However, he did not dictate the canonical trace, but allowed any trace consistent with a predicate characterizing the set of canonical traces. He required that no two canonical traces be equivalent and that every trace be equivalent to exactly one canonical trace.

The BDM canonical form for a black box is an abstraction of stimulus histories across all equivalence classes. This process of abstraction determines the basis for the future behavior of the black box for histories in each class. A BDM canonical form must have the following characteristics: (1) a single canonical form is specified for stimulus histories in all equivalence classes, (2) the form is tied to a basis ES, and (3) there must be an algorithm to map every stimulus history in the domain to a canonical stimulus history.

Definition For a black box f_b with SH, and an essential stimulus set ES, a *canonical form* is a set $CH \subseteq ES^*$ together with a function $Cn:SH \rightarrow CH$ defined so that for all $sh \in SH$, $Cn(sh) \equiv sh$. A stimulus history $ch \in CH$ is called a *canonical stimulus history*. Cn is called the *canonicalization function*.

A canonical form for a black box may not be unique. If an application is already in "simplest" terms, then $ES=VS$, $CH=SH$ and Cn the identity function. As with ES, it is an analysis and invention step of specification writing to choose a canonical form with the canonicalization function which will result in the "best" black box specification.

Definition For a black box f_b with SH and a canonical form (CH, Cn) , if for all $sh1, sh2 \in SH$, $[sh1 \equiv sh2 \rightarrow Cn(sh1) = Cn(sh2)]$, then (CH, Cn) is called a *unique canonicalization* of f_b .

In a unique canonicalization, each equivalence class has only one canonical stimulus history. All stimulus histories in the equivalence class can be transformed to this single canonical stimulus history. The example black boxes Stack and Birthday System of Appendix A have unique canonicalization.

Otherwise, (CH, Cn) performs a multiple canonicalization, where each equivalence class may have more than one canonical stimulus history. Not all stimulus histories in the equivalence class can be transformed by Cn to a single canonical stimulus history.

The known equivalences among stimulus histories become effective in developing a specification through the process of defining a specific canonical form (CH, Cn). In BDM, the equivalence relation on stimulus histories is implemented by an equivalence function defined as follows.

Definition For a black box f_b with VS and CH, Eq: $CH \times VS \rightarrow CH$, called the *equivalence*

function of f_b , maps canonical history ch concatenated with a stimulus s to its

equivalent new canonical history ch' , such that $ch' = Eq(ch, s) \wedge ch' \equiv ch+(s)$.

Once an equivalence function Eq is defined, an arbitrary stimulus history in the domain can be converted to a canonical history. A generic, BDM primitive, canonicalization function $Cn: SH \rightarrow CH$ is defined which maps a stimulus history sh to its canonical history ch based on the equivalence function. $Cn(sh) = Tr(\phi, sh)$, where Tr is also defined as a BDM primitive specification function to translate a ch concatenated with an arbitrary sh to its equivalent canonical history based on equivalence function Eq:

$$Tr(ch, sh) = [\#sh=0 \rightarrow ch \parallel \#sh \neq 0 \rightarrow Tr(Eq(ch, sh_1), sh_{2..s})]$$

Theorem (*Canonical Form Theorem*) For a black box f_b with SH, if Eq is an equivalence function with range $CH \subseteq SH$, then (CH, Cn) is a BDM canonical form.

Proof By the definition of an equivalence function and the definition of canonical form, to prove the theorem is to prove:

$$\forall s \in VS \forall ch \in CH [Eq(ch, s) \equiv ch+(s)] \Rightarrow \forall sh \in SH [Cn(sh) \equiv sh]$$

First, by induction on the length of sh:

$$\forall s \in VS \forall ch \in CH [Eq(ch, s) \equiv ch+(s)] \Rightarrow \forall ch \in CH \forall sh \in SH [Tr(ch, sh) \equiv ch+sh] \quad (1)$$

Basis: $\#sh=0$. Based on the definition of Tr, $ch = Tr(ch, \phi) \wedge ch \equiv ch+\phi$

Thus, the assertion (1) is true for the basis.

Induction step: Assume that assertion (1) is true for $\#sh < k$. Consider the case $\#sh = k$.

Based on the definition of Tr, $Tr(ch, sh_{1..k}) = Tr(Eq(ch, sh_1), sh_{2..k})$. (2)

Since $Eq(ch, sh_1) \in CH$ by the hypothesis, and $\#sh_{2..k} < k$, based on the assumption, we

have, $Tr(Eq(ch, sh_1), sh_{2..k}) \equiv Eq(ch, sh_1) + sh_{2..k}$ (3)

Since $Eq(ch, sh_1) \equiv ch + (sh_1)$ by hypothesis, $Eq(ch, sh_1) + sh_{2..k} \equiv ch + (sh_1) + sh_{2..k} = ch + sh_{1..k}$ (4)

Substitute $Tr(Eq(ch, sh_1), sh_{2..k})$ by $Tr(ch, sh_{1..k})$ based on (2), and substitute

$Eq(ch, sh_1) + sh_{2..k}$ by $ch + sh_{1..k}$ based on (4), into (3), to yield $Tr(ch, sh_{1..k}) \equiv ch + sh_{1..k}$

Therefore, assertion (1) also holds for the case $\#sh = k$.

Hence, assertion (1) is true and

$\forall s \in VS \forall ch \in CH [Eq(ch, s) \equiv ch + (s)] \Rightarrow \forall ch \in CH \forall sh \in SH [Tr(ch, sh) \equiv ch + sh],$

$\Rightarrow \forall sh \in SH [Tr(\phi, sh) \equiv \phi + sh], \Rightarrow \forall sh \in SH [Cn(sh) \equiv sh].$ □

Once a canonical form (CH, Cn) is defined for a black box f_b , we have $f_b(sh + \langle s \rangle) = f_b(Cn(sh) + \langle s \rangle) = f_b(ch + \langle s \rangle)$. Thus, the black box specification of f_b can be defined based on the CH instead of SH. Since the canonical stimulus history ch is in the "best" form possible for the application, the black box specification based on canonical stimulus histories is easier to write and read, which is the essence of behavior-based specification.

3.3 Black Box Specification

A black box is a specification for a software system or a component. A software specification usually contains a functional description and other behavioral aspects, such as response time, space efficiency, fault tolerance, security, and error recovery [5]. In this dissertation, the black box specification will focus on the description of functional behavior.

In the previous sections, the syntax and the semantics of well formed black boxes have been defined. However, the syntax and the semantics of WFBB do not guarantee a good

specification. The difficulties in constructing and verifying software specifications are similar to those encountered in program design. Due to the complexity of software systems, unstructured specifications, like unstructured programs, are hard to construct, read, and verify. A method similar to the structured programming can be applied to specification writing. The BDM *structured specification* method allows the specification of a complex system to be presented in a top-down, stepwise, functional style, which leads to readable, precise, complete and consistent hierarchical black box specifications.

A complex black box function may be defined in a hierarchy of black boxes, called a *specification hierarchy*, where higher-level black boxes describe the general structure of the system while lower-level black boxes describe more detailed mappings. Such a specification hierarchy decomposes the complexity to intellectually manageable units and supports synthesis of systems by reuse of existing black box specifications.

Domain partitioning allows the black box function to be presented as a union of a set of subfunctions corresponding to a response-based partition on the black box domain. Total case predicates and partial case predicates are introduced to facilitate domain partitioning and the mapping definition for piece-wise constructed functions.

A black box function description is based on a well formed mapping predicate. Although many well formed formulas could be used to define the characteristic function for a black box, they may not result in a good specification. BDM black boxes are constructed in structured forms with the goal of practical and accurate work.

3.3.1 Specification Hierarchy

In the original treatment of box structures, lower level black boxes do not occur until the clear box design, when the state box function is implemented by a procedure which organizes a set of lower level black boxes into the four basic control structures.

Since black boxes are mathematical functions, in BDM, black boxes are allowed to be used in a black box specification just as black boxes can be used in a clear box. The difference between them is the way in which these lower level black boxes are organized. A black box specification is a non-procedural description while a clear box is a procedure for a software system or a component. Therefore, subsidiary specification black boxes used in a black box specification are organized using other function symbols, relational operators and logical connectives into well formed formulas as a non-procedural, multi-level function definition. Such a hierarchy of specification black boxes used in a black box is called a *specification hierarchy* of a black box.

On the other hand, a regular black box is expanded to a state box and then a clear box. Such a BB-SB-CB design process is called *box expansion*. In a clear box design, the function is decomposed into a sequence use or a concurrent use of next level of black boxes. These next level black boxes will be expanded in turn. The hierarchy formed through such a top-down and stepwise box expansion is called a *usage hierarchy*, which will be discussed in chapter 5.

A specification hierarchy represents the decomposition of a complex black box function to a collection of smaller and less complex black boxes that are easier to write, read and verify. It also represents synthesis of systems by reuse of existing black boxes (i.e., the specifications of existing code units).

Lower level black boxes can be used in a black box specification in one of two ways, simple use or constructive use. Simple use of a lower level black box is a function call. For f_b , a subsidiary black box $\Delta g_b = ((S_1, S_2, \dots, S_m), (R_1, R_2, \dots, R_n), \delta, \chi)$ and any terms $x_1, x_2, \dots, x_m$, where $x_i \in S_i$, for $1 \leq i \leq m$, $g_b(x_1, x_2, \dots, x_m)$ can be used in $\Delta f_b \cdot \delta$ or $\Delta f_b \cdot \chi$ as a term. If g_b is used in case predicates in f_b , it can be either used in condition formulas or action formulas.

Constructive use of lower level black boxes is use of their definitions. In this way, a black box specification is defined by the components of the definitions of the lower level black boxes, rather than simply using them as terms. Two black box definitions are said to be *type compatible*, if each interface variable common to both black box interfaces has the identical type in each interface definition. Type compatibility is required for one black box to use another.

The simple use and the constructive use have different merit. Primitive black boxes and user defined common black boxes are used in the simple manner, since the arguments to two uses may contain different stimulus variables and response variables, and are not type-compatible.

Whereas, high-level specification of a system can be defined constructively using black box formulas containing subsystem definitions. In such cases, type compatibility among subsystems is usually satisfied. By using black box formulas, the components of the subsystem definitions do not need to be literally repeated in the definition of the whole system, as in the case of simple use.

Definition (Use Relation) Let B be a set of black boxes and let U be defined on B such that

$Uf_b g_b$ if and only if f_b uses g_b , i.e., g_b appears in Δf_b .

Definition (λ - Specification Hierarchy Function) For a black box f_b ,

$\lambda^1(\{f_b\}) = \{x \mid Uf_b x\}$, i.e., f_b and its direct uses of black boxes including itself.

$$\lambda^2(\{f_b\}) = \{ x \mid y \in \lambda^1(\{f_b\}) \wedge Uy, x \}$$

$$\lambda^k(\{f_b\}) = \{ x \mid y \in \lambda^{k-1}(\{f_b\}) \wedge Uy, x \}$$

$$\lambda^*(\{f_b\}) = \lambda^k(\{f_b\}) \text{ if } \lambda^k(\{f_b\}) = \lambda^{k+1}(\{f_b\})$$

$\lambda(\{f_b\})$ is the order-1 specification set of f_b , i.e., f_b and all the black boxes it uses directly. $\lambda^k(\{f_b\})$ is the order-k specification set of f_b with all black boxes it uses for k levels. λ^* refers to the complete specification set, or *specification closure*, regardless of the number of levels. A *specification hierarchy* is the tree representation of λ^* .

3.3.2 Domain Partitioning

Specification hierarchies are created by multi-level domain partitioning to represent a complex black box function by a set of subfunctions. Each partition gives rise to an equivalence relation on stimulus histories based on current response such that the response for stimulus histories in the same class can be described by the same subfunction.

Definition A *domain partitioning specification* of a black box f_b is a case predicate

$$C = [C_1 \rightarrow A_1 \parallel C_2 \rightarrow A_2 \parallel \dots \parallel C_n \rightarrow A_n] \text{ where for } 1 \leq i \leq n, SH/C = \{SH_1, SH_2, \dots, SH_n\}, \text{ with } SH_i = \{sh \mid sh \in SH \wedge C_i(sh)\}; \text{ where the action formula } A_i: SH_i \times R \rightarrow \{TRUE, FALSE\} \text{ determines the characteristic predicate for a subfunction } f_{ib}.$$

Theorem (*Domain Partitioning Theorem*) A case predicate $[C_1 \rightarrow A_1 \parallel C_2 \rightarrow A_2 \parallel \dots \parallel C_n \rightarrow A_n]$ forms a partition $SH_{p_1}, SH_{p_2}, \dots, SH_{p_n}$, where $\forall i \in \{1..n\} [SH_{p_i} = \{sh \mid sh \in SH_p \wedge C_i(sh)\}]$, on its domain SH_p if and only if

- (1) $\forall sh \in SH_p [(\bigvee_{i=1..n} C_i(sh)) = TRUE]$, and
- (2) $\forall i, j \in \{1..n\} \forall sh \in SH_p [i \neq j \Rightarrow \neg(C_i(sh) \wedge C_j(sh))]$.

Proof Based on the definition of partitioning, $SH_{p_1}, SH_{p_2}, \dots, SH_{p_n}$ form a partition if and only if (1) $\bigcup_{i=1..n} SH_{p_i} = SH_p$, and (2) $\forall i, j \in \{1..n\} [i \neq j \Rightarrow SH_{p_i} \cap SH_{p_j} = \emptyset]$.

$$\begin{aligned}
(1) \quad & \cup_{i=1..n} SH_{p_i} = SH_p \\
& \Leftrightarrow \{ sh \mid sh \in SH_p \wedge (C_1(sh) \vee C_2(sh) \vee \dots \vee C_n(sh)) \} = SH_p \\
& \Leftrightarrow \forall sh \in SH_p [(\vee_{i=1..n} C_i(sh)) = \text{TRUE}] \\
(2) \quad & \forall i, j \in \{1..n\} [i \neq j \Rightarrow SH_{p_i} \cap SH_{p_j} = \emptyset] \\
& \Leftrightarrow \forall i, j \in \{1..n\} [i \neq j \Rightarrow \{ sh \mid sh \in SH_p \wedge C_i(sh) \wedge C_j(sh) \} = \emptyset] \\
& \Leftrightarrow \forall i, j \in \{1..n\} [i \neq j \Rightarrow \forall sh \in SH_p (\neg(C_i(sh) \wedge C_j(sh)))] \\
& \Leftrightarrow \forall i, j \in \{1..n\} \forall sh \in SH_p [i \neq j \Rightarrow \neg(C_i(sh) \wedge C_j(sh))] \quad \square
\end{aligned}$$

Theorem (*Specification Assembly Theorem*) For black box f_b with SH and case predicate C, if C partitions SH, then $f_b = f_{1b} \cup f_{2b} \cup \dots \cup f_{nb}$.

Proof Let C be case predicate $[C_1 \rightarrow A_1 \parallel C_2 \rightarrow A_2 \parallel \dots \parallel C_n \rightarrow A_n]$ that forms a partition

$SH/C = \{SH_1, SH_2, \dots, SH_n\}$, where for $1 \leq i \leq n$, $SH_i = \{ sh \mid sh \in SH \wedge C_i(sh) \}$ and let

$f_{ib} = \{ (sh, r) \mid sh \in SH_i \wedge r \in R \wedge A_i(sh, r) \}$, for $1 \leq i \leq n$. Based on the semantics of

WFBB, we have

$$\begin{aligned}
f_b &= \{ (sh, r) \mid sh \in SH \wedge r \in R \wedge \chi(sh, r) \} \\
&= \{ (sh, r) \mid sh \in SH \wedge r \in R \wedge [C_1(sh) \rightarrow A_1(sh, r) \parallel C_2(sh) \rightarrow A_2(sh, r) \parallel \dots \parallel \\
&\quad C_n(sh) \rightarrow A_n(sh, r)] \} \\
&= \{ (sh, r) \mid (sh \in SH_1 \vee sh \in SH_2 \vee \dots \vee sh \in SH_n) \wedge r \in R \wedge ((C_1(sh)) \Rightarrow (A_1(sh, r))) \wedge \\
&\quad ((C_2(sh)) \Rightarrow (A_2(sh, r))) \wedge \dots \wedge ((C_n(sh)) \Rightarrow (A_n(sh, r))) \} \\
&= \{ (sh, r) \mid r \in R \wedge [sh \in SH_1 \wedge A_1(sh, r) \vee sh \in SH_2 \wedge A_2(sh, r) \vee \dots \vee sh \in SH_n \wedge A_n(sh, r)] \} \\
&= \{ (sh, r) \mid sh \in SH_1 \wedge r \in R \wedge A_1(sh, r) \} \cup \{ (sh, r) \mid sh \in SH_2 \wedge r \in R \wedge A_2(sh, r) \} \cup \\
&\quad \dots \cup \{ (sh, r) \mid sh \in SH_n \wedge r \in R \wedge A_n(sh, r) \} \\
&= f_{1b} \cup f_{2b} \cup \dots \cup f_{nb} \quad \square
\end{aligned}$$

Case predicates are used to decompose the complexity of a black box. For a very complex system, one level of domain partition may not be sufficient. Case predicates can be nested in action formulas to form a multi-level domain partitioning.

The following terminology is used in reference to the relationship among case predicates used in black box specifications. Let $P = [C_1 \rightarrow A_1 \parallel \dots \parallel C_m \rightarrow A_m]$, $T = [D_1 \rightarrow B_1 \parallel \dots \parallel D_n \rightarrow B_n]$ be case predicates, and W be an arbitrary well-formed formula: In formula $[C_1 \rightarrow A_1 \parallel \dots \parallel C_i \rightarrow A_i \wedge T \wedge A_{i_2} \parallel \dots \parallel C_m \rightarrow A_m]$, P is called the *parent* case predicate of T , and T is the *child* case predicate of P ; In formula $P \wedge T$ or $T \wedge P$, P and T are *sibling* case predicates; In formula $W \wedge T$ or $T \wedge W$, W is said to be a *sibling* predicate of case predicate T ; When two case predicates have either parent-child relation or sibling-sibling relation, the two are said to be *adjacent*; The case predicate that has no parent case predicate is called the *root case predicate*.

Definition The *domain of a case predicate* T , denoted SH_T , is defined inductively as follows:

- (1) if T is the root case predicate, the domain of T is the domain SH of the black box,
- (2) if T is a child of a case predicate P which has a domain SH_P and T appears in one of the action formulas of P such that $[C_1 \rightarrow A_1 \parallel \dots \parallel C_i \rightarrow A_i \wedge T \wedge A_{i_2} \parallel \dots \parallel C_m \rightarrow A_m]$, then, the domain of T is $\{ sh \mid sh \in SH_P \wedge C_i(sh) \}$.

3.3.3 Normal Black Boxes

Even though one could choose to represent a black box mapping by many different WFF's, BDM recommends a standard form of specification structures for the representation of the mappings based on using multi-level domain partitions and a specification hierarchy. Such a structured representation not only provides a well structured specification, but also supports a

systematic verification of the consistency and completeness of the specification, which will be discussed below.

Definition A black box is a *normal black box* if its mapping predicate χ is in the form

$A_1 \wedge \dots \wedge A_n$, for $n \geq 1$, where each A_i is either a domain partitioning specification or a (quantified) relational expression such that (1) there is no case predicate used in any condition formula, and (2) every action formula in any case predicate is in the form $B_1 \wedge \dots \wedge B_m$, for $m \geq 1$, where each B_i is either a domain partitioning specification or a (quantified) relational expression.

Definition A normal black box is *restricted normal* if in its mapping predicate χ no case predicate has a sibling predicate in terms of the $\wedge$ operation. A restricted normal black box is called *depth-restricted normal* if only zero or one case predicate is used.

The following theorem shows that no loss is suffered by restricting software specifications to normal black boxes.

Theorem Any partial recursive function can be defined as a normal black box function.

Proof (1) Basic recursive functions: successor function $S(x)=x+1$, zero function $N(x)=0$, and generalized identity function $U_i^{(n)}(x_1, x_2, \dots, x_n)=x_i$ can be defined in BDM as normal black boxes:

BB $S(\text{ in } x, \text{ out Return }) \{ \text{ Return } = x+1 \}$

BB $N(\text{ in } x, \text{ out Return }) \{ \text{ Return } = 0 \}$

BB $U(\text{ in } i, x_1, x_2, \dots, x_n, \text{ out Return }) \{ \text{ Return } = x_i \}$

(2) Function composition: given functions $f(y_1, y_2, \dots, y_n)$, $g_1(x_1, x_2, \dots, x_m)$,

$g_2(x_1, x_2, \dots, x_m), \dots, g_n(x_1, x_2, \dots, x_m)$, function composition $f(g_1(x_1, x_2, \dots, x_m),$

$g_2(x_1, x_2, \dots, x_m), \dots, g_n(x_1, x_2, \dots, x_m))$ is a valid term allowed in normal black boxes.

(3) Primitive recursion: given primitive recursive functions $g(x_1, x_2, \dots, x_{n-1})$, $h(x_1, x_2, \dots, x_{n+1})$, a primitive recursion $f(x_1, x_2, \dots, x_n)$ can be defined as normal black box:

BB $f(\text{in } x_1, x_2, \dots, x_n, \text{out Return})$
 $\{ [x_n=0 \rightarrow \text{Return} = g(x_1, x_2, \dots, x_{n-1})$
 $\parallel \text{otherwise} \rightarrow \text{Return} = h(x_1, x_2, \dots, x_n, f(x_1, x_2, \dots, x_{n-1}, x_{n-1}))] \}$

(4) Minimization: for a given $f(y, x_1, x_2, \dots, x_n)$, the minimization whose value is the least y such that $f(y, x_1, x_2, \dots, x_n)=0$ for given $(x_1, x_2, \dots, x_n)$ can be defined as $\text{Min}(\{ y \mid f(y, x_1, x_2, \dots, x_n)=0 \})$ and used as a term in normal black boxes.

(5) Thus, any partial recursive function, which can be obtained by repeated application of composition, primitive recursion and minimization starting with the successor, zero and generalized identity functions, can be define as a normal black box by repeated application of the above BDM representation of composition, primitive recursion, minimization, successor, zero and the generalized identity functions. $\square$

Theorem (*Predicate Join*) Let $P = [C_1 \rightarrow A_1 \parallel \dots \parallel C_i \rightarrow A_i \parallel \dots \parallel C_m \rightarrow A_m]$ and $T = [D_1 \rightarrow B_1 \parallel \dots \parallel D_n \rightarrow B_n]$ be two sibling case predicates that each forms a partition on their common domain SH. Then, $P \wedge T = ([C_1 \rightarrow A_1 \parallel \dots \parallel C_m \rightarrow A_m] \wedge [D_1 \rightarrow B_1 \parallel \dots \parallel D_n \rightarrow B_n])$
 $\Leftrightarrow PT = [C_1 \wedge D_1 \rightarrow A_1 \wedge B_1 \parallel \dots \parallel C_1 \wedge D_n \rightarrow A_1 \wedge B_n$
 $\parallel C_2 \wedge D_1 \rightarrow A_2 \wedge B_1 \parallel \dots \parallel C_2 \wedge D_n \rightarrow A_2 \wedge B_n \dots$
 $\parallel C_m \wedge D_1 \rightarrow A_m \wedge B_1 \parallel \dots \parallel C_m \wedge D_n \rightarrow A_m \wedge B_n]$

Proof Since P and T each form a partition on their domain, according to the domain partitioning theorem, for any stimulus history in the domain SH,
 $\exists i \in \{1..m\} (C_i = \text{TRUE})$ and $\exists j \in \{1..n\} (D_j = \text{TRUE})$. Thus, every case in P is vacuously true (since its hypothesis is false) except $C_i \rightarrow A_i$, every case in T is vacuously true

except $D_j \rightarrow B_j$, and every case in PT is vacuously true except $C_i \wedge D_j \rightarrow A_i \wedge B_j$.

Therefore, for any sh such that $C_i(\text{sh})$ is true and $D_j(\text{sh})$ is true,

$$\begin{aligned}
& [C_i \rightarrow A_i \parallel \dots \parallel C_m \rightarrow A_m] \wedge [D_1 \rightarrow B_1 \parallel \dots \parallel D_n \rightarrow B_n] \\
\Leftrightarrow & (C_i \Rightarrow A_i) \wedge (D_j \Rightarrow B_j) \\
\Leftrightarrow & (\text{TRUE} \Rightarrow A_i) \wedge (\text{TRUE} \Rightarrow B_j) \\
\Leftrightarrow & A_i \wedge B_j \\
\Leftrightarrow & (\text{TRUE} \Rightarrow A_i \wedge B_j) \\
\Leftrightarrow & (C_i \wedge D_j \Rightarrow A_i \wedge B_j) \\
\Leftrightarrow & [C_i \wedge D_1 \rightarrow A_i \wedge B_1 \parallel \dots \parallel C_i \wedge D_n \rightarrow A_i \wedge B_n \\
& \parallel C_2 \wedge D_1 \rightarrow A_2 \wedge B_1 \parallel \dots \parallel C_2 \wedge D_n \rightarrow A_2 \wedge B_n \\
& \dots \\
& \parallel C_m \wedge D_1 \rightarrow A_m \wedge B_1 \parallel \dots \parallel C_m \wedge D_n \rightarrow A_m \wedge B_n]
\end{aligned}$$

Since sh, i, and j are arbitrary, the equivalence is proved true. $\square$

Theorem (*Predicate Absorption*) Let $P = [C_i \rightarrow A_i \parallel \dots \parallel C_i \rightarrow T \parallel \dots \parallel C_m \rightarrow A_m]$ and

$T = [D_1 \rightarrow B_1 \parallel \dots \parallel D_n \rightarrow B_n]$ be two case predicates in a parent-child relationship. Then

T can be absorbed into P such that:

$$\begin{aligned}
& [C_i \rightarrow A_i \parallel \dots \parallel C_i \rightarrow [D_1 \rightarrow B_1 \parallel \dots \parallel D_n \rightarrow B_n] \parallel \dots \parallel C_m \rightarrow A_m] \\
\Leftrightarrow & [C_i \rightarrow A_i \parallel \dots \parallel C_i \wedge D_1 \rightarrow B_1 \parallel C_i \wedge D_2 \rightarrow B_2 \parallel \dots \parallel C_i \wedge D_n \rightarrow B_n \parallel \dots \parallel C_m \rightarrow A_m]
\end{aligned}$$

Proof $C_i \rightarrow [D_1 \rightarrow B_1 \parallel \dots \parallel D_n \rightarrow B_n]$

$$\begin{aligned}
\Leftrightarrow & \neg C_i \vee [(\neg D_1 \vee B_1) \wedge \dots \wedge (\neg D_n \vee B_n)] \\
\Leftrightarrow & (\neg C_i \vee \neg D_1 \vee B_1) \wedge \dots \wedge (\neg C_i \vee \neg D_n \vee B_n) \\
\Leftrightarrow & (\neg(C_i \wedge D_1) \vee B_1) \wedge \dots \wedge (\neg(C_i \wedge D_n) \vee B_n) \\
\Leftrightarrow & [C_i \wedge D_1 \rightarrow B_1 \parallel \dots \parallel C_i \wedge D_n \rightarrow B_n]
\end{aligned}$$

The theorem follows. $\square$

Theorem (Action Distribution) If $P = [C_1 \rightarrow A_1 \parallel \dots \parallel C_i \rightarrow A_i \parallel \dots \parallel C_m \rightarrow A_m]$ forms a partition

on its domain SH_P and there is no variable in any condition formula C_i , for $1 \leq i \leq m$,

that is constrained by some equation in a wff B that contains no case predicate, then

$$([C_1 \rightarrow A_1 \parallel \dots \parallel C_m \rightarrow A_m] \wedge B) \Leftrightarrow [C_1 \rightarrow A_1 \wedge B \parallel \dots \parallel C_m \rightarrow A_m \wedge B]$$

Proof This is a special case of predicate join. Since there is no variable in any condition

formula C_i , for $1 \leq i \leq m$, that is constrained by B , and $B \Leftrightarrow [TRUE \rightarrow B]$,

$$([C_1 \rightarrow A_1 \parallel \dots \parallel C_m \rightarrow A_m] \wedge B)$$

$$\Leftrightarrow ([C_1 \rightarrow A_1 \parallel \dots \parallel C_m \rightarrow A_m] \wedge [TRUE \rightarrow B])$$

$$\Leftrightarrow [C_1 \rightarrow A_1 \wedge B \parallel \dots \parallel C_m \rightarrow A_m \wedge B] \text{ by the predicate join rule.} \quad \square$$

Theorem (Join Partition) Let $P = [C_1 \rightarrow A_1 \parallel \dots \parallel C_m \rightarrow A_m]$ and $T = [D_1 \rightarrow B_1 \parallel \dots \parallel D_n \rightarrow B_n]$

be two sibling case predicates. Let

$$PT = [C_1 \wedge D_1 \rightarrow A_1 \wedge B_1 \parallel \dots \parallel C_1 \wedge D_n \rightarrow A_1 \wedge B_n$$

$$\parallel C_2 \wedge D_1 \rightarrow A_2 \wedge B_1 \parallel \dots \parallel C_2 \wedge D_n \rightarrow A_2 \wedge B_n$$

..

$$\parallel C_m \wedge D_1 \rightarrow A_m \wedge B_1 \parallel \dots \parallel C_m \wedge D_n \rightarrow A_m \wedge B_n].$$

Then, PT forms a partition on its domain if and only if both P and T form partitions

on their domains.

Proof Let SH_P be the domain of P , T and PT . Let $E_k = C_i \wedge D_j$ and $F_k = A_i \wedge B_j$, for $1 \leq i \leq m$,

$1 \leq j \leq n$, and $1 \leq k \leq mn$. Then, PT becomes

$$PT = [E_1 \rightarrow F_1 \parallel E_2 \rightarrow F_2 \parallel \dots \parallel E_{mn} \rightarrow F_{mn}]$$

By the Exactly One Theorem,

P and T form partitions on their domains

$$\Leftrightarrow \forall sh \in SH_P \exists i \in \{1..m\} [C_i(sh)] \wedge \forall sh \in SH_P \exists j \in \{1..n\} [D_j(sh)]$$

$$\Leftrightarrow \forall sh \in SH_P \exists i \in \{1..m\} \exists j \in \{1..n\} [C_i(sh) \wedge D_j(sh)]$$

$$\Leftrightarrow \forall sh \in SH_p \exists k \in \{1..mn\} [E_k]$$

$$\Leftrightarrow PT \text{ forms partitions on its domain}$$

□

Theorem (*Absorption Partition*) Let $P = [C_1 \rightarrow A_1 \parallel .. \parallel C_l \rightarrow T \parallel .. \parallel C_m \rightarrow A_m]$ and

$T = [D_1 \rightarrow B_1 \parallel .. \parallel D_n \rightarrow B_n]$ be two case predicates parent-child related. Let PT be the case predicate that results from predicate absorption. PT forms a partition on its domain if and only if both P and T form partitions on their respective domains.

Proof Let SH_p be the domain of P, $SH_T = \{sh | sh \in SH_p \wedge C_l(sh)\}$ be the domain of T, and

$$\begin{aligned} PT = & [C_1 \rightarrow A_1 \parallel .. \parallel C_{l-1} \rightarrow A_{l-1} \\ & \parallel C_l \wedge D_1 \rightarrow B_1 \parallel C_l \wedge D_2 \rightarrow B_2 \parallel .. \parallel C_l \wedge D_n \rightarrow B_n \\ & \parallel C_{l+1} \rightarrow A_{l+1} \parallel .. \parallel C_m \rightarrow A_m] \end{aligned}$$

By the Domain Partitioning Theorem, two assertions must be proved.

(1) The disjunction of all condition formulas of PT is TRUE if and only if the disjunction of all condition formulas of P is TRUE and the disjunction of all condition formulas of T is TRUE.

$$\begin{aligned} \forall h \in SH_p \forall h' \in SH_T [& (C_1(h) \vee .. \vee C_{l-1}(h) \vee \\ & C_l(h) \wedge (D_1(h') \vee D_2(h') \vee .. \vee D_n(h')) \vee \\ & C_{l+1}(h) \vee .. \vee C_m(h)) = \text{TRUE}] \end{aligned}$$

$$\Leftrightarrow \forall h \in SH_p [(\vee_{i=1,m} C_i(h)) = \text{TRUE}] \wedge \forall h' \in SH_T [C_l(h') \Rightarrow (\vee_{j=1,n} D_j(h')) = \text{TRUE}]$$

$$\Rightarrow \forall h \in SH_p [(\vee_{i=1,m} C_i(h)) = \text{TRUE}] \wedge \forall h' \in SH_T [(\vee_{j=1,n} D_j(h')) = \text{TRUE}]$$

(2) The conjunction of any two condition formulas of PT is FALSE if and only if the conjunction of any two condition formulas of P is FALSE and the conjunction of any two condition formulas of T is FALSE. The if-part is a straight implication, i.e.,

$$\forall i, j \in \{1..m\} \forall h \in SH_p [i \neq j \Rightarrow \neg(C_i(h) \wedge C_j(h))] \wedge$$

$$\forall i, j \in \{1..n\} \forall h' \in SH_T [i \neq j \Rightarrow \neg(D_i(h') \wedge D_j(h'))]$$

$$\begin{aligned}
\Rightarrow & \quad \forall i, j \in \{1, \dots, l-1, l+1, \dots, m\} \forall h \in SH_p[i \neq j \Rightarrow \neg(C_i(h) \wedge C_j(h))] \wedge \\
& \quad \forall i \in \{1, \dots, l-1, l+1, \dots, m\} \forall j \in \{1..n\} \forall h \in SH_p \forall h' \in SH_T[\neg(C_i(h) \wedge C_l(h) \wedge D_j(h'))] \wedge \\
& \quad \forall j, k \in \{1..n\} \forall h \in SH_p \forall h' \in SH_T[j \neq k \Rightarrow \neg(C_l(h) \wedge D_j(h') \wedge D_k(h'))]
\end{aligned}$$

Only if part:

$$\begin{aligned}
& \neg (\forall i, j \in \{1..m\} \forall h \in SH_p[i \neq j \Rightarrow \neg(C_i(h) \wedge C_j(h))] \wedge \\
& \quad \forall i, j \in \{1..n\} \forall h' \in SH_T[i \neq j \Rightarrow \neg(D_i(h') \wedge D_j(h'))]) \\
\Leftrightarrow & \quad \exists i, j \in \{1..m\} \exists h \in SH_p[i \neq j \wedge C_i(h) \wedge C_j(h)] \vee \\
& \quad \exists i, j \in \{1..n\} \exists h' \in SH_T[i \neq j \wedge D_i(h') \wedge D_j(h')] \\
\Rightarrow & \quad \exists i, j \in \{1, \dots, l-1, l+1, \dots, m\} \exists h \in SH_p[i \neq j \wedge C_i(h) \wedge C_j(h)] \vee \\
& \quad \exists i \in \{1, \dots, l-1, l+1, \dots, m\} \exists j \in \{1..n\} \exists h \in SH_p \exists h' \in SH_T[\neg(C_i(h) \wedge C_l(h) \wedge D_j(h'))] \vee \\
& \quad \exists j, k \in \{1..n\} \exists h \in SH_p \exists h' \in SH_T[j \neq k \wedge C_l(h) \wedge D_j(h') \wedge D_k(h')] \\
\Leftrightarrow & \quad \neg (\forall i, j \in \{1, \dots, l-1, l+1, \dots, m\} \forall h \in SH_p[i \neq j \Rightarrow \neg(C_i(h) \wedge C_j(h))] \wedge \\
& \quad \forall i \in \{1, \dots, l-1, l+1, \dots, m\} \forall j \in \{1..n\} \forall h \in SH_p \forall h' \in SH_T[\neg(C_i(h) \wedge C_l(h) \wedge D_j(h'))] \wedge \\
& \quad \forall j, k \in \{1..n\} \forall h \in SH_p \forall h' \in SH_T[j \neq k \Rightarrow \neg(C_l(h) \wedge D_j(h') \wedge D_k(h'))]) \quad \square
\end{aligned}$$

Thus, in a normal black box containing nested case predicates, adjacent case predicates can be merged and action formulas can be distributed into case predicates. By such merging and distribution, the normal black box can be transformed to a depth-restricted normal black box as shown in the following theorem.

Theorem (*Depth Restriction of Normal Black Box*) For any normal black box f_b , there exists an equivalent depth-restricted normal black box f'_b .

Proof Let $\Delta f_b = (SL, RL, \delta, \chi)$. Prove by induction on the number of case predicates in f_b .

Basis: For $N = 0$, the mapping predicate χ of f_b is in the form $A_1 \wedge \dots \wedge A_n$, for $n \geq 1$,

where each A_i is a (quantified) relational expression. Then, the equivalent restricted

normal black box $\Delta f'_b = (SL, RL, \delta, [TRUE \rightarrow A_1 \wedge \dots \wedge A_n])$ is a depth-restricted normal black box using only one case predicate. Thus, the assertion holds for $N=0$. For $N = 1$, if χ itself is a case predicate, then f_b is already a depth-restricted normal black box. Otherwise, χ must be in the form $A_1 \wedge \dots \wedge A_n$, for $n \geq 1$, where A_1 is a case predicate $[C_1 \rightarrow B_1 \parallel \dots \parallel C_m \rightarrow B_m]$ and the other A_i , for $2 \leq i \leq n$, are (quantified) relational expressions, then χ can be transformed to an equivalent case predicate $\chi' =$

$$[C'_1 \rightarrow B_1 \wedge A_2 \wedge \dots \wedge A_n \parallel C'_2 \rightarrow B_2 \wedge A_2 \wedge \dots \wedge A_n \parallel \dots \parallel C'_m \rightarrow B_m \wedge A_2 \wedge \dots \wedge A_n]$$

where for $1 \leq j \leq m$, each C'_j is the result of repeatedly substituting every variable y in C_j , that is constrained by an equation $y = g(x_1, \dots, x_k)$ for some function g in $A_2 \wedge \dots \wedge A_n$, by $g(x_1, \dots, x_k)$ until there is no variable in C'_j is constrained by $A_2 \wedge \dots \wedge A_n$.

According to the Substitution Theorem [43], The value of each C'_j is the same as the value of each C_j . Thus, χ forms a partition on SH if and only if χ' forms a partition on SH. Thus, $\Delta f'_b = (SL, RL, \delta, \chi')$ is an equivalent depth-restricted normal black box using just one case predicate. The assertion holds for $N=1$.

Induction step: Assume the assertion holds for $N < K$, where $K > 1$. Consider the case $N=K$. Suppose there are K case predicates in the mapping predicate χ of f_b . Let $P = [C_1 \rightarrow A_1 \parallel \dots \parallel C_m \rightarrow A_m]$ and $T = [D_1 \rightarrow B_1 \parallel \dots \parallel D_n \rightarrow B_n]$ be two arbitrary adjacent case predicates in χ . If P and T are sibling predicates, i.e., $P \wedge T$ is a subformula in χ , then by the Predicate Join and Join Partition Theorems, $P \wedge T$ can be merged to one case predicate PT , and PT forms a partition. Thus, the number of case predicates in χ will be less than K so that the assertion holds. If P and T are in a parent-child relationship, i.e., $[C_1 \rightarrow A_1 \parallel \dots \parallel C_i \rightarrow T \wedge A'_i \parallel \dots \parallel C_m \rightarrow A_m]$ where $A_i = T \wedge A'_i$ is a subformula in χ , then, by the Predicate Absorption and Absorption Partition Theorems, P and T can be merged into one case predicate PT , and PT forms a partition. Thus, the number of

case predicates in T will be less than K so that the assertion also holds. Therefore, the assertion also holds for the case $N=K$. □

Therefore, any normal black box has an equivalent depth-restricted normal black box. However, a normal black box, which contains case predicates in parent-child and sibling relationships and having common action subformula abstractions, fewer expressions and providing a multi-level presentation of a complex black box function will generally produce a better specification than will a depth-restricted normal black box. The depth-restricted normal black box concept is useful mainly for study of mathematical properties of black boxes.

3.3.4 SH-based Specification

Direct *SH-based specification*, is an approach in which the black box mapping is defined directly in terms of stimulus histories and their subhistories. In a SH-based specification, concepts, entities, and their relationships in an application domain are formally defined as sets, lists, predicates and functions as building blocks using BDM list notation, operations, and primitive functions. The black box mapping is described using these building blocks in the form of a normal or restricted normal black box. Given a problem statement, the following steps will lead to a SH-based specification.

Requirement analysis

Requirement analysis is a discovery process to identify concepts, data entities, and operations in the application domain from the user requirement. Usually, requirement analysis consists of data analysis and function analysis.

A. Data analysis. Identify data entities, their names, structures, value ranges and semantics. Record each data entity in a *data analysis table* where each entry contains four

components (*Data, Structure, Range, Semantics*). For example, data analysis of the problem statement for the Birthday Book identified the following data entities.

<u>Data</u>	<u>Structure</u>	<u>Range</u>	<u>Semantics</u>
name	character string		
birthday	(month, day, year)	1/1/1900 < birthday	
person	(name, birthday)		
known	person*		persons entered and not subsequently deleted

B. Function analysis. A black box function may be a union of a set of subfunctions. Each subfunction is called an operation or a transaction. Function analysis is to identify every operation, and the stimulus data entities, the conditions (the relationship between the current stimulus and the prior stimulus history), and the response data entities that are involved in the operation. Record each operation in a *function analysis table* where each entry contains four components (*Operation, Input, Condition, Output*). Continuing the example, function analysis of the problem statement for the Birthday Book identified the following operations.

<u>Operation</u>	<u>Input</u>	<u>Condition</u>	<u>Output</u>
Add	(name, birthday)	name not added before name added before	
Delete	name	name not added before name added before	
Find	name	name not added before name added before	birthday
Remind	birthday		list of names

The above requirement analysis revealed that the problem statement is not complete. Responses for some stimulus histories, such as a stimulus history ending with an add request, and a stimulus history ending with a delete request, are not defined. Although responses for such stimulus histories could be null, a specific response other than a null would be a better response to various stimulus histories. An incomplete user requirement needs to be revised by interacting with sponsors, or by making assumptions until it is complete. A function analysis table for Birthday Book after the requirement is completed could be like the following table:

<u>Operation</u>	<u>Input</u>	<u>Condition</u>	<u>Output</u>
Add	(name, birthday)	name not added before	"Add confirmation"
		name added before	"Name already known"
Delete	name	name not added before	"Name not known"
		name added before	"Delete confirmation"
Find	name	name not added before	"Name not known"
		name added before	birthday
Remind	birthday		list of names

Specification

Specification writing is a creative process to establish a formal model in BDM, based on the results of the requirements analysis, that defines the function of the software system.

A. Data definition. Data entities in the application domain discovered in data analysis are abstracted and classified into abstract data types using ADT statements or concrete data types using typedef statements. For a large system, such abstract types and concrete types constitute a *data dictionary*. Since data definition is an abstraction step, there may not be a one-to-one correspondence between the data entities identified in data analysis and the data

types and variables defined in data definition. In the Birthday Book example, both abstract and concrete types for NAME, DATE and NAMELIST are defined.

B. Interface definition. (1) Define stimulus variables and their types based on abstraction from the union of input data entities appearing in the input column of the function analysis table. Any value range imposed on data entities in data analysis that correspond to stimulus variable will be defined by the domain predicate δ . For a black box with multiple operations, a specific operation type will be selected for each use of the box. Thus, operation type is usually a component of a stimulus. In the Birthday Book example, based on an abstraction from the union of input data entities appearing in the input column of the function analysis table, three stimulus variables are defined: Command as a stimulus type for operation type, InName as a stimulus type for input names, and InDate as a stimulus type for input birthday. (2) Define response variables and their types, based on abstraction from the union of output data entities appearing in the output column of the function analysis table. In the Birthday Book example, based on an abstraction from the union of output data entities appearing in the output column of the function analysis table, three response variables are defined: Message as responses for an Add or a Delete request, NameList as a response for a Remind request, and OutDate as a response for a Find request.

C. Stimulus history dependency definition. Each response is a function of a stimulus history. But sometimes, a response may only depend on a critical sublist of a stimulus history instead of the whole history itself. Each response and its dependency on stimulus histories are examined based on the condition column in the function analysis table. Sublists of stimulus histories that determine responses are identified and formally defined as named lists in *def .. for* clause, using BDM list notation and operations to facilitate the description of black box mapping. In the Birthday Book example, the condition column of the

function analysis table contains conditions about whether a name is added to the system before. Thus, Known, a sublist of a stimulus history, is defined for persons whose names were previously added and were not subsequently deleted and can be used to decide whether each condition is true or not for a stimulus history.

D. Mapping definition. The black box mapping predicate is defined in terms of stimulus histories and their named sublists defined in step C. For a black box with multiple operations, the domain is partitioned using case predicate based on operation types identified in the function analysis table. In the SH-based black box Birthday Book, the mapping predicate is a domain partitioning specification on the operation types ADD, DELETE, FIND, and REMIND. The sublist Known is used to determine responses.

SH-based approach is a transaction-oriented specification rather than a property-oriented description. Such a specification is usually direct, simple, and close to the original problem statement. A disadvantage of the direct SH-based specification is that when the response dependency on stimulus histories and the relationship among stimuli in stimulus histories are complex, a direct SH-based specification is difficult to achieve.

Example

A. An SH-based black box for MinMaxAvg12 can be found in Appendix A. In the next section a black box definition in the CH-based approach for MaxMinAvg12 is given. It will show that the SH-based approach yields a much better specification for MaxMinAvg12 than a CH-based specification.

B. An SH-based black box for the Birthday Book can be found in Appendix A. In the SH-based black box BirthdayBook, the mapping predicate is an English-like and very readable yet precise specification. Each predicate symbol contained in the mapping χ is precisely defined in the symbol definition section. Each predicate symbol represents a concept

or an entity in the application domain. For examples, in the stimulus history, same name may appear more than once with different birthdays. Which one will be used to produce responses to stimuli FIND or REMIND? It is not necessarily the first pair of the name and the birthday, since the first pair may be deleted by a DELETE command appearing later in the stimulus history. The precise semantics is pretty hard to express in natural English. In BDM, such semantics can be precisely defined without losing the readability. In the example, Known is defined as a sublist of the stimulus history and represents the concept of a set of (name, birthday) tuples whose "names were previously added and were not subsequently deleted". Compared to the Z specification [24], such a SH-based black box BirthdayBook in BDM is more concise and simpler. Compared to its CH-based specification that will be presented next, the SH-based specification is direct and closer to the user's English description of the application concepts, hence more readable.

3.3.5 CH-based Specification

For some applications, the response dependencies on stimulus histories and the relationship among stimuli in stimulus histories are complex. Thus, a direct SH-based specification can be difficult to write. A *CH-based specification* is an approach in which a BDM canonical form is defined first so that every stimulus history can be converted to a canonical history, and then the mapping predicate is defined in terms of the canonical stimulus histories. Since a canonical stimulus history is usually simpler than the general stimulus history, the black box mapping will be easier to describe in terms of the canonical history. Given a problem statement, the following steps will lead to a CH-based black box specification.

Requirement analysis

Data analysis and function analysis of user requirement are the same as that of a SH-based specification.

Specification

A. Data definition is the same as that of a SH-based specification.

B. Interface definition is the same as that of a SH-based specification.

C. Analysis of effect of stimuli on future behavior. The impact of possible stimuli on future behavior is analyzed based on the function analysis table. From the set of all valid stimuli VS, first identify the set of omissible stimuli, OS, those having no effect on future behavior. Next, identify within VS-OS the set(s) ES which can represent the semantics of any stimulus history in SH. Stimuli that are not omissible and are not in ES are those replaceable by a string from ES^* , form a set denoted RS. There may be more than one possible ES and a judgment will have to be made as to which will result in the best specification. For example, an examination on the function analysis table for the Birthday Book revealed that future behavior of the black box depends on the persons whose names were previously added and were not subsequently deleted. The function of a REMIND request or a FIND request has nothing to do with such persons, thus, they have no impact on future behaviors. Therefore, $OS = \{(FIND, *, \epsilon), (REMIND, \epsilon, *)\}$. Then the remaining VS-OS can be divided as $ES = \{(ADD, *, *)\}$ and $RS = \{(DELETE, *, \epsilon)\}$.

D. Equivalence function definition. Define the equivalence function

$Eq: CH \times VS \rightarrow CH$ by defining a domain predicate for the canonical form and a mapping predicate for an equivalence relation such that for a given old canonical history och and for

each possible stimulus s that follows och , the new canonical history nch is defined so that $nch = Eq(och, s) \wedge nch \equiv och + \langle s \rangle$. Eq will have the following general form

BB $Eq(in\ och, s, out\ Return)$

```
{  dom  och  $\in$  <canonical-form-definition>  $\wedge$  s  $\in$  S
  map  def nch=Return; for
    [ s $\in$ OS  $\rightarrow$  nch = och
      || s $\in$ ES  $\vee$  s $\in$ RS  $\rightarrow$ 
        [ C1(och, s)  $\rightarrow$  nch = Eq1(och, s)
          || C2(och, s)  $\rightarrow$  nch = Eq2(och, s)
          || ..
          || Cn(och, s)  $\rightarrow$  nch = Eqn(och, s) ] ]
}
```

where, OS, RS, and ES are defined in step C, $\langle\text{canonical-form-definition}\rangle \subseteq ES^*$, and Eq_i for $1 \leq i \leq n$ will be defined based on the impact of the specific stimulus to the future behavior implied in the function analysis table. For the Birthday Book example, since $ES = \{ (ADD, *, *) \}$, a canonical form definition can be $CH = (\{ADD\} \times NAME \times DATE)^*$. The canonical form of a stimulus history is its sublist of (ADD, InName, InDate) stimuli where each name was not added before and was not subsequently deleted. According to the function analysis table, for a stimulus (ADD, Name, *), Name will be added only if it has not been added before. And for a stimulus (DELETE, Name, ϵ), Name will be deleted only if it has been added before. Thus, an Eq for black box Birthday Book can be defined as follows.

BB Eq(in och, s, out Return)

```
{  dom  och ∈ ({ADD} × NAME × DATE)* ∧ s ∈ COMMAND × NAME × DATE
    map def nch = Return; for
        [ s = (ADD, Name, *) ∧ (ADD, Name, *) ∉ och      →  nch = och + ⟨s⟩
          || s = (DELETE, Name, *) ∧ (ADD, Name, *) ∈ och →  nch = och - ⟨(ADD, Name, *)⟩
          || otherwise                                     →  nch = och ]
}
```

E. Mapping definition. The black box mapping predicate is defined in terms of canonical stimulus history. For a black box with multiple operations, the domain is partitioned using case predicate based on operation types identified in the function analysis table. In the CH-based black box Birthday Book, the mapping predicate is a domain partitioning specification on the operation types ADD, DELETE, FIND, and REMIND. Known is defined as the canonical form of the prior stimulus history and is used to determine responses. Complete CH-based black boxes for Stack, BirthdayBook, and MinMaxAvg12 can be found in Appendix A.

The advantages of a CH-based specification are that for complex stimulus histories, the CH-based approach will make the stimulus history analysis and access simpler; and CH and its equivalence relation are close to state encapsulation and state maintenance. Thus, a CH-based black box specification will make state box analysis and definition easier. The disadvantage of such a specification is that it might tend to obscure the essential simplicity of black boxes.

Summary Advice

- (1) Use normal black boxes.
- (2) Use SH-based specification approach when practical.

- (3) If SH-based approach is not practical, use CH-based specification approach.
- (4) If CH-based approach is not practical, complete the parts of the black box description that are practical, defer or omit the impractical details and go on to a state-based specification using a state box.

3.4 Black Box Verification

A formal specification possesses certain mathematical properties. Two important properties of interest are completeness and consistency. In this section, the concepts of completeness and consistency are defined. Theorems which provide a method of verification of consistency and completeness of normal black boxes are presented.

3.4.1 Completeness and Consistency

A complete black box specification defines a response for every valid input history. It fully determines the function to be implemented. The behavior of the specified system is so fully defined that no implementation that has more or less behavior can be constructed [23]. If a black box specification should represent a relation that is not a function, $f_b(sh)$ represents an arbitrary one of the possible responses.

Definition A black box specification f_b is *syntactically complete* if and only if

$$\forall sh[\Delta f_b \vdash (sh \in SH) \Rightarrow \exists r \in R[\Delta f_b \vdash (f_b(sh)=r)]]$$

A black box specification is consistent if it does not permit contradictory derivations such as $P \wedge \neg P$ from some input. In other words, for each valid stimulus history, the response of the black box should always be the same.

Syntactical consistency is formally defined as follows.

Definition A black box specification f_b is *syntactically consistent* if and only if

$$\forall sh[\Delta f_b \vdash (sh \in SH) \Rightarrow \neg \exists r \in R[\Delta f_b \vdash (f_b(sh)=r \wedge f_b(sh) \neq r)]]$$

3.4.2 Specification Verification Theorems

BDM black boxes are verified for syntactical consistency and syntactical completeness. The verification theorems stated below can be used for normal black box specifications. The central ideas behind the verification theorems are (1) to prove completeness and consistency by verifying a finite number of subfunctions defined by domain partitioning; and (2) to establish these properties for a large and complex black box by proving them for a collection of smaller and less complex (lower level) black boxes.

Lemma For any wff $D_1, D_2, \dots, D_k$,

$$(\bigvee_{i=1,k} D_i) = \text{TRUE} \wedge \forall i,j \in \{1..k\} [i \neq j \Rightarrow \neg(D_i \wedge D_j)] \text{ if and only if } \exists i \in \{1..k\} [D_i]$$

Proof Since $\exists i \in \{1..k\} [D_i] \Leftrightarrow$

$$[(D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_k) \vee (\neg D_1 \wedge D_2 \wedge \neg D_3 \wedge \dots \wedge \neg D_k) \vee \dots \vee (\neg D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_{k-1} \wedge D_k)],$$

we prove the assertion by proving

$$(\bigvee_{i=1,k} D_i) = \text{TRUE} \wedge \forall i,j \in \{1..k\} [i \neq j \Rightarrow \neg(D_i \wedge D_j)] \Leftrightarrow$$

$$[(D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_k) \vee (\neg D_1 \wedge D_2 \wedge \neg D_3 \wedge \dots \wedge \neg D_k) \vee \dots \vee (\neg D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_{k-1} \wedge D_k)].$$

Prove by induction on k . Basis: For $k=2$,

$$(\bigvee_{i=1,2} D_i) = \text{TRUE} \wedge \forall i,j \in \{1,2\} [i \neq j \Rightarrow \neg(D_i \wedge D_j)] \Leftrightarrow (D_1 \vee D_2) \wedge \neg(D_1 \wedge D_2) \Leftrightarrow$$

$$(D_1 \vee D_2) \wedge (\neg D_1 \vee \neg D_2) \Leftrightarrow (D_1 \wedge \neg D_2) \vee (\neg D_1 \wedge D_2). \text{ Thus, the assertion holds for } k=2.$$

Induction step: Assume the assertion is true for $k=n$, for $n \geq 2$. Consider the case

$k=n+1$. Let $\forall i \in \{1..n-1\} [D'_i = D_i]$ and $D'_n = D_n \vee D_{n+1}$. Then,

$$(\bigvee_{i=1,n+1} D_i) = \text{TRUE} \wedge \forall i,j \in \{1..n+1\} [i \neq j \Rightarrow \neg(D_i \wedge D_j)]$$

$$\Leftrightarrow (\bigvee_{i=1,n+1} D_i) = \text{TRUE} \wedge \forall i,j \in \{1..n-1\} [i \neq j \Rightarrow \neg(D_i \wedge D_j)] \wedge$$

$$\begin{aligned}
& \forall i \in \{1..n-1\} [\neg(D_i \wedge (D_n \vee D_{n+1}))] \wedge \neg(D_n \wedge D_{n+1}) \\
\Leftrightarrow & (\bigvee_{i=1..n} D'_i) = \text{TRUE} \wedge \forall i, j \in \{1..n-1\} [i \neq j \Rightarrow \neg(D'_i \wedge D'_j)] \wedge \\
& \forall i \in \{1..n-1\} [\neg(D'_i \wedge D'_n)] \wedge \neg(D_n \wedge D_{n+1}) \\
\Leftrightarrow & (\bigvee_{i=1..n} D'_i) = \text{TRUE} \wedge \forall i, j \in \{1..n\} [i \neq j \Rightarrow \neg(D'_i \wedge D'_j)] \wedge \neg(D_n \wedge D_{n+1}) \\
\Leftrightarrow & [(D'_1 \wedge \neg D'_2 \wedge \dots \wedge \neg D'_n) \vee (\neg D'_1 \wedge D'_2 \wedge \dots \wedge \neg D'_n) \vee \dots \vee \\
& (\neg D'_1 \wedge \neg D'_2 \wedge \dots \wedge \neg D'_{n-1} \wedge D'_n)] \wedge \neg(D_n \wedge D_{n+1}) \\
\Leftrightarrow & [D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_n \wedge \neg D_{n+1} \wedge \neg(D_n \wedge D_{n+1}) \vee \\
& \neg D_1 \wedge D_2 \wedge \neg D_3 \wedge \dots \wedge \neg D_n \wedge \neg D_{n+1} \wedge \neg(D_n \wedge D_{n+1}) \vee \dots \vee \\
& \neg D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_{n-1} \wedge (D_n \vee D_{n+1}) \wedge \neg(D_n \wedge D_{n+1})] \\
\Leftrightarrow & [D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_n \wedge \neg D_{n+1} \vee \\
& \neg D_1 \wedge D_2 \wedge \dots \wedge \neg D_n \wedge \neg D_{n+1} \vee \dots \vee \\
& \neg D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_{n-1} \wedge (D_n \wedge \neg D_{n+1} \vee \neg D_n \wedge D_{n+1})] \\
\Leftrightarrow & [D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_n \wedge \neg D_{n+1} \vee \\
& \neg D_1 \wedge D_2 \wedge \dots \wedge \neg D_n \wedge \neg D_{n+1} \vee \dots \vee \\
& \neg D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_{n-1} \wedge D_n \wedge \neg D_{n+1} \vee \\
& \neg D_1 \wedge \neg D_2 \wedge \dots \wedge \neg D_{n-1} \wedge \neg D_n \wedge D_{n+1})]
\end{aligned}$$

Therefore, the assertion also holds for $k=n+1$. □

Theorem (Exactly One) For any black box f_b with domain SH, and a case predicate

$[C_1 \rightarrow A_1 \parallel \dots \parallel C_k \rightarrow A_k]$ used in Δf_b having domain SH_p

$\forall sh \in SH_p [(\bigvee_{i=1..k} C_i(sh)) = \text{TRUE} \wedge \forall i, j \in \{1..k\} [i \neq j \Rightarrow \neg(C_i(sh) \wedge C_j(sh))]]$

if and only if $\forall sh \in SH_p \exists i \in \{1..k\} [C_i(sh)]$

Proof The result follows immediately from substituting quantified wff's $C_1, \dots, C_k$ for the arbitrary wff's $D_1, \dots, D_k$ in the lemma. □

Theorem (*Completeness and Consistency*) A black box specification is complete and

consistent if and only if $\forall sh[\Delta f_b \vdash (sh \in SH) \Rightarrow \exists_1 r \in R[\Delta f_b \vdash (f_b(sh)=r)]]$

Proof Since a black box specification is complete if and only if

$\forall sh[\Delta f_b \vdash (sh \in SH) \Rightarrow \exists r \in R[\Delta f_b \vdash (f_b(sh)=r)]]$, and a black box specification is consistent

if and only if $\forall sh[\Delta f_b \vdash (sh \in SH) \Rightarrow \neg \exists r \in R[\Delta f_b \vdash (f_b(sh)=r \wedge f_b(sh) \neq r)]]$, thus, a black

box specification is complete and consistent if and only if

$\forall sh[\Delta f_b \vdash (sh \in SH) \Rightarrow \exists r \in R[\Delta f_b \vdash (f_b(sh)=r)]] \wedge$

$\forall sh[\Delta f_b \vdash (sh \in SH) \Rightarrow \neg \exists r \in R[\Delta f_b \vdash (f_b(sh)=r \wedge f_b(sh) \neq r)]]$

$\Leftrightarrow \forall sh[\Delta f_b \vdash (sh \in SH) \Rightarrow \exists r \in R[\Delta f_b \vdash (f_b(sh)=r)] \wedge \neg \exists r \in R[\Delta f_b \vdash (f_b(sh)=r \wedge f_b(sh) \neq r)]]$

$\Leftrightarrow \forall sh[\Delta f_b \vdash (sh \in SH) \Rightarrow \exists_1 r \in R[\Delta f_b \vdash (f_b(sh)=r)]]$ □

A depth-restricted normal black box is complete and consistent if and only if every action formula for each case defines a unique final state for each output variable, as stated in the following theorem.

Theorem (*Verification of Depth-restricted Normal Black Box*) A depth-restricted normal

black box $\Delta f_b = ((S_1, \dots, S_m), (R_1, \dots, R_n), \delta, \chi)$, where $\chi = [C_1 \rightarrow A_1 \parallel \dots \parallel C_k \rightarrow A_k]$, is

complete and consistent if and only if in its case predicate

$\forall sh \in SH \forall i \in \{1..k\} [C_i(sh) \Rightarrow \exists_1 r \in R[A_i(sh, r)]]$, i.e., in each action formula, the final state of each output variable R_i of RL is uniquely defined.

Proof If part: By the Exactly One Theorem, $\forall sh \in SH [\exists_1 i \in \{1..k\} (C_i(sh))]$. Thus,

$\forall sh \in SH [\exists_1 i \in \{1..k\} (C_i(sh))] \wedge \forall sh \in SH \forall i \in \{1..k\} [C_i(sh) \Rightarrow \exists_1 r \in R[A_i(sh, r)]]$

$\Rightarrow \forall sh \in SH \exists_1 i \in \{1..k\} \exists_1 r \in R[A_i(sh, r)]$

$\Rightarrow \forall sh[\Delta f_b \vdash (sh \in SH) \Rightarrow \exists_1 r \in R[\Delta f_b \vdash (f_b(sh)=r)]]$,

by Domain Partitioning Specification

$\Leftrightarrow \Delta f_b$ is complete and consistent, by the Completeness and Consistency Theorem.

Only if part:

$$\begin{aligned}
& \neg \forall sh \in SH \forall i \in \{1..k\} [C_i(sh) \Rightarrow \exists_1 r \in R [A_i(sh, r)]] \\
& \Leftrightarrow \exists sh \in SH \exists i \in \{1..k\} [C_i(sh) \wedge \neg \exists_1 r \in R [A_i(sh, r)]] \\
& \Rightarrow \exists sh \in SH \neg \exists_1 r \in R [\Delta f_b \vdash (r = f_b(sh))] \\
& \Rightarrow \neg \forall sh [\Delta f_b \vdash (sh \in SH) \Rightarrow \exists_1 r \in R [\Delta f_b \vdash (f_b(sh) = r)]] \\
& \Leftrightarrow \Delta f_b \text{ is not complete and consistent,}
\end{aligned}$$

by Completeness and Consistency Theorem. □

Theorem (*Verification of Restricted Normal Black Box*) A restricted normal black box Δf_b is complete and consistent if and only if in every action formula of each case predicate used in Δf_b , the final state of each output variable in RL is uniquely defined.

Proof By the Depth-restriction of Normal Black Box Theorem, the restricted normal black box Δf_b can be transformed to an equivalent depth-restricted normal black box $\Delta f'_b$ such that $f_b = f'_b$. By the definition of restricted normal black box, there is no sibling relationship among case predicates used in Δf_b . Thus, in transforming Δf_b to $\Delta f'_b$, only predicate absorption will occur. Thus, all action formulas of the case predicates used in Δf_b will be preserved in the case predicate of $\Delta f'_b$. Thus, Δf_b satisfies the condition here if and only if $\Delta f'_b$ satisfies the condition in the Depth-restricted Verification Theorem.

Since $f_b = f'_b$, Δf_b is complete and consistent

$$\begin{aligned}
& \Leftrightarrow \Delta f'_b \text{ is complete and consistent} \\
& \Leftrightarrow \Delta f'_b \text{ satisfies the condition in Depth-restricted Verification} \\
& \Leftrightarrow \Delta f_b \text{ satisfies the condition here.}
\end{aligned}$$
□

Theorem (*Verification of Normal Black Box*) The completeness and the consistency of a normal black box specification Δf_b can be verified by: (1) Transforming Δf_b to an

equivalent restricted normal black box $\Delta f'_b$ by using Action Distribution and Predicate Join Theorems to eliminate sibling case predicates, (2) Δf_b is complete and consistent if and only if $\Delta f'_b$ is complete and consistent which can be determined by the Restricted Normal Verification Theorem.

Proof Requirement (1) can be proved by induction on the number of case predicates used in Δf_b , similar to the proof of the Depth-restriction of Normal Black Box Theorem. (2) follows from $f_b = f'_b$. □

When the above three verification theorems are applied to a simple black box, the verification conditions can be decided within the black box. However, if the verification theorems are applied to a composite black box, the verification conditions are satisfied only if all lower-level black boxes are complete and consistent within their application domains, the set of all possible stimulus histories they actually receive in the context of their applications.

Black Box Verification Method

For a black box specification Δf_b , its completeness and consistency can be verified as follows:

(1) Verify that Δf_b is a normal black box by

(a) Verify the specification structure.

(b) Verify that every case predicate used in Δf_b forms a partition on its domain.

(2) If Δf_b is normal, then use the verification theorems.

Example

Verify the completeness and consistency of the CH-based black box BirthdayBook defined in Appendix A. It is a composite black box using a BDM primitive function Cn and a user defined equivalence function Eq. It is a restricted normal black box. According to the Restricted Normal Verification Theorem, we verify the black box BirthdayBook as follows:

(1) Eq is complete and consistent since the case predicate used in it forms a partition on its domain and each action formula uniquely defines nch.

(2) Cn is complete and consistent since both Eq and the primitive function Tr are complete and consistent.

(3) Every case predicate used in $\Delta\text{BirthdayBook}.\chi$ forms a partition on its domain.

Although each action formula does not explicitly define a unique value for every response variable, in this case they are Message, OutDate, and NameList, the partial implication operator " $\dashv$ " implies a unique value has been defined for every response variable. For example, if the current stimulus is (DELETE, John Smith, *) and John Smith is known to the system based on the prior stimulus history, then according to the semantics of " $\dashv$ " and the initial state of pure response variables OutDate^{*} and NameList^{*} are ϵ , we have

$$\begin{aligned} & [\Delta\text{BirthdayBook}.\chi \wedge (\text{Command}, \text{InName}, \text{InDate}) = (\text{DELETE}, \text{John Smith}, *) \\ & \quad \wedge \text{AlreadyKnown}(\text{John Smith})] \dashv \text{Message} = \text{"Delete confirmation"} \\ \Leftrightarrow & [\Delta\text{BirthdayBook}.\chi \wedge (\text{Command}, \text{InName}, \text{InDate}) = (\text{DELETE}, \text{John Smith}, *) \\ & \quad \wedge \text{AlreadyKnown}(\text{John Smith})] \rightarrow \text{Message} = \text{"Delete confirmation"} \wedge \\ & \quad \text{OutDate} = \epsilon \wedge \text{NameList} = \epsilon \end{aligned}$$

Thus, for this specific stimulus history, the action formula does define a unique value for every response variable. Therefore, the CH-based black box BirthdayBook is complete and consistent. If in $\Delta\text{BirthdayBook}.\chi$, all " $\dashv$ " are replaced by " $\rightarrow$ ", then it will not be complete and consistent, since each action formula will not define a unique value for every response variable.

Chapter 4

BDM State Boxes

This chapter presents the semantics of well formed state boxes, state encapsulation of stimulus histories, state migration, state box design and state box verification.

In BSM, a state box is an intermediate system view that defines internal state. The state box has the identical external behavior as its black box.

The classical state box is a state machine with states introduced to capture essential information from stimulus histories and a machine defined to carry out the state transitions and the response productions, so that the response is determined by only the current stimulus and the current state as described by: $(\text{Stimulus, Old State}) \rightarrow (\text{Response, New State})$. However, implementations of software systems typically contain multiple levels of data abstractions, with state data decentralized and migrated to the smallest system parts that do not require duplicating data updates [2]. The classical state box does not model state migration, so Mills proposed a "generalized" state box to accommodate state migration with $g:S^* \times T^* \rightarrow R \times T$, where T is the set of states, instead of the classical $g:S \times T \rightarrow R \times T$ [3].

The BDM state box has a significantly different definition from those given by Mills, but is formulated to model the same phenomenon. Moreover, the BDM goal is to illuminate the task faced by the box structures practitioner and provide guidance in state box design and state migration. The definition of a state box is tied to the definition of the black box from which it is expanded. (In Chapter 5, the definition of a clear box is tied to the definition of the state box from which it is expanded). Some details are spelled out regarding the

implications of state migration being specified during state box (versus during clear box) design.

The classical state box appears as a special case of the generalized form. Top (high) level state boxes typically do not address state migration, i.e., they migrate all state information or none of it; those that migrate none of it are of this form. When state migration has been specified using generalized state boxes, at the final design level state boxes will be of the classical form. While the generalized form is required for the intermediate design steps involving state migration, its mathematical complexity has clouded the essential role and compelling simplicity of the classical state box. As will be seen, most of the theoretical help developed here is associated with the classical state box, making a strong case for state migration to be decided at the clear box level rather than specified during state box design.

A state box is a state-defined, procedure-free description of its black box function. However, the state box does constrain clear box design because it must be viewed as the specification for the clear box. If one of the (generally several possible) classical state boxes is chosen, the clear box design is relatively unconstrained. As a state-based specification, a state box shares some characteristics and properties with a black box specification. Some specification and verification concepts described in Chapter 3, such as domain partitioning, specification hierarchy, normal specification and verification theorems, also apply to state boxes.

A state verification method is introduced which is conducted directly on the original black box and the invented state box as an alternative to the derive-and-compare technique of BSM.

4.1 State Box Semantics

Definition A well formed state box, denoted f_s , has definition denoted Δf_s , and

$\Delta f_s = (\Delta f_b, (QL, q_1, \Delta f_{sb}), \text{ where } \Delta f_b \text{ is WFBB,}$

$QL = (tQ_1 Q_1, tQ_2 Q_2, \dots, tQ_k Q_k)$ is a finite list of state variable declarations,

q_1 is the *initial state* of $Q \subseteq t(Q_1)_\varepsilon \times t(Q_2)_\varepsilon \times \dots \times t(Q_k)_\varepsilon$, the set of all possible

states, $\Delta f_{sb} = (QL + \Delta f_b.SL, QL + \Delta f_b.RL, \delta_{sb}, \chi_{sb})$ is a WFBB, and $f_s = f_b$.

The definition of the state box includes the definition of the black box from which it was expanded and which dictates its external behavior, a state space to be invented and an internal WFBB to be invented. Commitment to a state space constrains, but does not determine, the clear box. The domain predicate of the internal black box $\delta_{sb}: (Q \times S)^+ \rightarrow \{\text{TRUE}, \text{FALSE}\}$ determines the domain of the partial function $\chi_{sb}: (Q \times S)^+ \times (Q \times R) \rightarrow \{\text{TRUE}, \text{FALSE}\}$ and its function f_{sb} yields a response mapping function $f_R = \pi_R f_{sb}$ and a state mapping function $f_Q = \pi_Q f_{sb}$. Thus, the external behavior of state box function f_s must be recovered by the behavior of a pair of functions (f_R, f_Q) used to construct f_{sb} on domain $\{ \text{qsh} \mid \text{qsh} \in (Q \times S)^+ \wedge \delta_b(\pi_S(\text{qsh})) \wedge \forall i \in \{2.. \# \text{qsh}\} [\pi_Q(\text{qsh}_i) = \pi_Q f_{sb}(\text{qsh}_{1..i-1})] \}$.

$f_s = \{ (sh, r) \mid sh = \langle s_1, \dots, s_n \rangle \wedge sh \in SH \wedge r \in R \wedge$

$\exists \text{qsh} \in (Q \times S)^n [\pi_S(\text{qsh}) = sh \wedge \forall i \in \{2..n\} [\pi_Q(\text{qsh}_i) = f_Q(\text{qsh}_{1..i-1})] \wedge r = f_R(\text{qsh})] \}$

If the internal black box f_{sb} produces each response based on only the current stimulus and the internal state, f_{sb} becomes a mapping $f_{sb}: Q \times S \rightarrow Q \times R$ and f_s is a *classical state box* such that:

$f_s = \{ (sh, r) \mid sh = \langle s_1, \dots, s_n \rangle \wedge sh \in SH \wedge r \in R \wedge \forall i \in \{1..n\} [q_{i+1} = f_Q(q_i, s_i)] \wedge r = f_R(q_n, s_n) \}$

A black box potentially has many correct state boxes; many different state spaces and initial states are possible, and for each state space and initial state many internal black boxes are possible, as may be easily illustrated. BDM emphasizes a "design path" where the parts of

one box are used in its expansion according to the definitions given. In this chapter and the next, a binary relation β (_ is the box expansion of _) on the set of all boxes (black, state and clear) is defined in parts.

Definition f_s is a box expansion of f_b , denoted $\beta(f_b, f_s)$, if Δf_b is used in Δf_s .

4.2 State Encapsulation

The state space Q of a state box represents an encapsulation of stimulus histories of the black box. Given a black box function, many different Q can be defined as the encapsulation of the stimulus histories. The expansion of a black box to one or more state boxes represents choice among alternative design tradeoffs. The design of a state box is a human invention step and no general algorithm can be given that will satisfactorily encapsulate the state for an arbitrary black box, independently of design criteria and objectives.

Definition Given $\beta(f_b, f_s)$, a function $\sigma: S^* \rightarrow Q$ that maps black box stimulus histories to states is called an *encapsulation function* and can be defined recursively as follows:

$$\sigma(\phi) = q_1$$

$$\sigma(sh) = f_Q(qsh) = f_Q(\langle\langle\sigma(\phi), s_1\rangle, \langle\sigma(\langle s_1 \rangle), s_2\rangle, \dots, \langle\sigma(\langle s_1, s_2, \dots, s_{n-1} \rangle), s_n\rangle\rangle).$$

In the case that state box f_s is a classical state box, σ can be simplified to:

$$\sigma(\phi) = q_1 \text{ and } \sigma(sh) = f_Q(q_n, s_n) = f_Q(\langle\langle s_1, s_2, \dots, s_{n-1} \rangle\rangle, s_n).$$

For $\beta(f_b, f_s)$, $h = \langle s_1, \dots, s_{n-1} \rangle$ and encapsulation function σ , $f_b(h + \langle s_n \rangle) = f_s(h + \langle s_n \rangle) = f_R(\langle\langle q_1, s_1 \rangle, \langle q_2, s_2 \rangle, \dots, \langle q_n, s_n \rangle \rangle) = f_R(\langle\langle q_1, s_1 \rangle, \langle \sigma(h_{1..1}), s_2 \rangle, \dots, \langle \sigma(h_{1..n-1}), s_n \rangle \rangle)$. If f_s is a classical state box, σ encapsulates $h_{1..n-1}$ to a current state q_n by repeated application of f_Q such that $q_n = \sigma(h) = f_Q(\dots(f_Q(f_Q(q_1, s_1), s_2), \dots, s_{n-1}), s_n)$, we have, $f_b(h + \langle s_n \rangle) = f_s(h + \langle s_n \rangle) = f_R(q_n, s_n) = f_R(\sigma(h), s_n) = f_R(f_Q(\dots(f_Q(f_Q(q_1, s_1), s_2), \dots, s_{n-1}), s_n))$

Thus the identical external view is preserved when $\beta(f_b, f_s)$. However, the state box function f_s is represented by f_R and f_Q which depend on only the current stimulus and current state. These expressions do not suggest that stimulus histories are still needed for the state box. On the contrary, the power of a state box lies in the fact that stimulus histories are encapsulated into the state space incrementally, such that $f_b(h+\langle s_n \rangle) = f_s(h+\langle s_n \rangle) = f_R(q_n, s_n)$, and $q_{n+1} = f_Q(q_n, s_n)$. This demonstrates both preservation of the external views of a black box and any of its state boxes, and the relationship between the external view and the internal view of a state box.

In a generalized state box, the state does not encapsulate all the information contained in a stimulus history and the state-stimulus history is required. However, in the special case of a classical state box, a state does encapsulate all the information needed for producing future responses.

Definition If $\beta(f_b, f_s)$ and f_s is a classical state box, f_s has a *state trace* function $f_Q^*: Q \times S^* \rightarrow Q$

with $\text{dom}(f_Q^*) = \{ (q, h) \mid q \in Q \wedge h \in S^* \wedge (q, h_1) \in \text{dom}(f_Q) \wedge q_2 = f_Q(q, h_1) \wedge$

$\forall i \in \{2.. \#h\} [(q_i, h_i) \in \text{dom}(f_Q) \wedge q_{i+1} = f_Q(q_i, h_i)] \}$

and $f_Q^*(q, \phi) = q$, $f_Q^*(q, h) = f_Q(f_Q^*(q, h_{1..s-1}), h_s)$.

The encapsulation function is related to the state trace function as follows:

$f_Q^*(q, h) = f_Q(..(f_Q(f_Q(q, h_1), h_2), .., h_{s-1}), h_s)$, and $\sigma(sh) = f_Q^*(q_1, sh)$.

Definition If $\beta(f_b, f_s)$ and f_s is a classical state box, two states q and q' are *equivalent* with

respect to $f_{sb}: Q \times S \rightarrow Q \times R$, denoted $q \equiv q'$, if and only if $\forall h \in S^* [\text{Defined}(q, h) =$

$\text{Defined}(q', h) \wedge (\text{Defined}(q, h) \Rightarrow f_R(f_Q^*(q, h_{1..s-1}), h_s) = f_R(f_Q^*(q', h_{1..s-1}), h_s))]$, where the

predicate $\text{Defined}(q, h) = [(q, h_{1..s-1}) \in \text{dom}(f_Q^*) \wedge (f_Q^*(q, h_{1..s-1}), h_s) \in \text{dom}(f_R)]$.

Definition If $\beta(f_b, f_s)$ and f_s is a classical state box, Q is a *minimal state space* if and only if

$\forall q, q' \in Q [q \equiv q' \Rightarrow q = q']$.

Definition If $\beta(f_b, f_s)$ and $\forall h+\langle s \rangle \in SH \exists q \in Q [q = \sigma(h) \wedge f_b(h+\langle s \rangle) = f_s(\langle q, s \rangle)]$, then Q is a *total state space*.

Definition If a state space is both total and minimal, then it is an *optimal state space*.

Definition Let $\beta(f_b, f_{s1}), \beta(f_b, f_{s2})$, and let $\Delta f_{s1}.Q$ be optimal. If $\Delta f_{s2}.Q \subset \Delta f_{s1}.Q$, then $\Delta f_{s1}.Q$ is *subset-migrated state*. If $\Delta f_{s2}.QL \subset \Delta f_{s1}.QL$ and $\Delta f_{s2}.Q = \pi_{\Delta f_{s2}.QL}(\Delta f_{s1}.Q)$, then $\pi_{\Delta f_{s1}.QL - \Delta f_{s2}.QL}(\Delta f_{s1}.Q)$ is *component-migrated state*. State space that is not migrated (by subset-migration or component-migration) is *retained state space*.

Theorem (State Equivalence) If $\beta(f_b, f_s)$ and f_s is a classical state box,

$$\forall sh1, sh2 \in SH [sh1 \equiv sh2 \Leftrightarrow \sigma(sh1) \equiv \sigma(sh2)].$$

Proof For any $sh1, sh2 \in SH$, let $q = \sigma(sh1)$ and $q' = \sigma(sh2)$. Then, we have,

$$sh1 \equiv sh2$$

$$\Leftrightarrow \forall h \in S^+ [(sh1+h \in SH) = (sh2+h \in SH) \wedge (sh1+h \in SH \Rightarrow f_b(sh1+h) = f_b(sh2+h))]$$

$$\Leftrightarrow \forall h \in S^+ [((\sigma(sh1+h_{1..s-1}), h_s) \in \text{dom}(f_{sb})) = ((\sigma(sh2+h_{1..s-1}), h_s) \in \text{dom}(f_{sb})) \wedge$$

$$((\sigma(sh1+h_{1..s-1}), h_s) \in \text{dom}(f_{sb}) \Rightarrow f_R(\sigma(sh1+h_{1..s-1}), h_s) = f_R(\sigma(sh2+h_{1..s-1}), h_s))]$$

$$\Leftrightarrow \forall h \in S^+ [\text{Defined}(q_1, sh1+h) = \text{Defined}(q_1, sh2+h) \wedge$$

$$(\text{Defined}(q_1, sh1+h) \Rightarrow f_R(f_Q^*(q_1, sh1+h_{1..s-1}), h_s) = f_R(f_Q^*(q_1, sh2+h_{1..s-1}), h_s))]$$

$$\Leftrightarrow \forall h \in S^+ [q = \sigma(sh1) \wedge q' = \sigma(sh2) \wedge \text{Defined}(q, h) = \text{Defined}(q', h) \wedge$$

$$(\text{Defined}(q, h) \Rightarrow f_R(f_Q^*(q, h_{1..s-1}), h_s) = f_R(f_Q^*(q', h_{1..s-1}), h_s))]$$

$$\Leftrightarrow \sigma(sh1) \equiv \sigma(sh2)$$

□

Theorem (Canonical History Encapsulation) Let f_b be a black box with canonical form (CH, Cn), then there exists a state box f_s such that $\beta(f_b, f_s)$ and $Q = CH$ and $\sigma = Cn$.

Proof Let $\Delta f_b = (SL, RL, \delta, \chi_b)$ where the CH-based description χ_b is

{def BB Eq(..){..}; ch=Cn(h); for w}, where $w: (CH \times S) \times R \rightarrow \{TRUE, FALSE\}$ is a wff.

Regarding set CH as a type and an element of the set as an instance of the type,

$QL=(CH \ q)$ and $Q=CH$, thus Cn conforms to the requirements for a state encapsulation function, and $\sigma=Cn$. Define $q_1 = \sigma(\phi) = Cn(\phi) = \phi$.

The internal f_{sb} is defined by χ_{sb} as a pair of functions ($f_Q:Q \times S \rightarrow Q, f_R:Q \times S \rightarrow R$).

By the definition of Eq,

$$\forall ch \in CH \forall s \in S [Eq(ch,s) \equiv ch + \langle s \rangle]$$

$$\Rightarrow \forall ch \in CH \forall s \in S [\sigma(Eq(ch,s)) \equiv \sigma(ch + \langle s \rangle)], \text{ by State Equivalence Theorem}$$

$$\Rightarrow \forall ch \in CH \forall s \in S [Cn(Eq(ch,s)) \equiv f_Q(\sigma(ch),s)]$$

$$\Rightarrow \forall ch \in CH \forall s \in S [Eq(ch,s) \equiv f_Q(ch,s)]. \text{ Thus, define } f_Q=Eq.$$

By the definition of WFBB and WFSB,

$$\forall ch \in CH \forall s \in S [f_b(ch + \langle s \rangle) = f_R(\sigma(ch), s) = f_R(ch, s). \text{ Thus, define } f_R(q,s) = f_b(q + \langle s \rangle).$$

Therefore, $\Delta f_s = (\Delta f_b, (\langle CH \ q \rangle, \phi), \Delta f_{sb})$, where $\Delta f_{sb} = (\langle \langle CH \ q \rangle \rangle + \Delta f_b, SL, \langle \langle CH$

$q \rangle \rangle + \Delta f_b, RL, \Delta f_b, \delta(q + \langle s \rangle), \chi_{sb})$, satisfies the requirements of the theorem. $\square$

Theorem (Null State) If $\forall h + \langle s \rangle \in SH [f_b(h + \langle s \rangle) = f_b(\langle s \rangle)]$, then $\Delta f_s = (\Delta f_b, (\phi, \epsilon), \Delta f_{sb})$ has optimal state space $\{\epsilon\}$.

Proof Define $\forall s \in S [f_Q(\epsilon, s) = \epsilon \wedge f_R(\epsilon, s) = f_b(\langle s \rangle)]$, then

$$\forall h + \langle s \rangle \in SH [\sigma(h) = \epsilon \wedge f_b(h + \langle s \rangle) = f_R(\epsilon, s) = f_b(\langle s \rangle)]. \text{ Thus, } \{\epsilon\} \text{ is a total state space,}$$

$\{\epsilon\}$ is trivially minimal, and $\{\epsilon\}$ is optimal. $\square$

Theorem (Minimized State Space)

If $\beta(f_b, f_s)$, where f_b has a unique canonicalization (CH, Cn) and f_s is a classical state box, then $Q' = \sigma(CH) \subseteq \Delta f_s.Q$ is a minimal state space.

Proof Assume the assertion is false, i.e., for such a $\beta(f_b, f_s)$, the Q' is not minimal. Thus,

$$\exists q, q' \in Q' [q \equiv q' \wedge q \neq q']$$

$$\Rightarrow \exists ch1, ch2 \in CH [q = \sigma(ch1) \wedge q' = \sigma(ch2) \wedge ch1 \equiv ch2 \wedge ch1 \neq ch2],$$

by the State Equivalence Theorem, but this is a contradiction of the hypothesis that (CH, Cn) is a unique canonicalization. $\square$

Two algebras will be constructed, a CH-equivalence algebra and Q-transition algebra, and a homomorphism will be established from the equivalence algebra to the transition algebra. This will permit properties of one algebra to be derived from properties of the other algebra [46] which creates a potential mechanism for black box design advice to flow into state box design.

Definition If Δf_b is a black box with canonical form (CH, Cn) , its *CH-equivalence algebra* is

$\mathbb{E}_b = (CH \times S, Eq', (\phi, \epsilon))$, where

- 1) S is the set of all possible stimuli including ϵ ,
- 2) CH is the set of canonical histories,
- 3) Eq' is a unary operator such that $Eq'(ch, s) = (Eq(ch, s), s)$, and
- 4) (ϕ, ϵ) is a constant such that $Eq'(\phi, \epsilon) = (\phi, \epsilon)$.

Definition If Δf_s is a classical state box, its *Q-transition algebra* is $T_s = (Q \times S, f_Q', (q_1, \epsilon))$,

where

- 1) S is the set of all possible stimuli including ϵ ,
- 2) Q and q_1 are state space and the initial state, respectively,
- 3) f_Q' is a unary operator such that $f_Q'(q, s) = (f_Q(q, s), s)$, where f_Q is from f_{sb} and
- 4) (q_1, ϵ) is a constant such that $f_Q'(q_1, \epsilon) = (q_1, \epsilon)$.

Theorem (*ET Homomorphism*) If $\beta(f_b, f_s)$ and f_s is a classical state box with minimal state space, and $\sigma': CH \times S \rightarrow Q \times S$ is defined by $\sigma'(ch, s) = (\sigma(ch), s)$, then σ' is a homomorphism from $\mathbb{E}_b$ to T_s .

Proof Based on the definition of Eq ,

$$Eq(ch, s) \equiv ch + \langle s \rangle$$

$$\begin{aligned}
&\Rightarrow \sigma(\text{Eq}(\text{ch}, s)) \equiv \sigma(\text{ch} + \langle s \rangle), && \text{by State Equivalence Theorem} \\
&\Rightarrow \sigma(\text{Eq}(\text{ch}, s)) = \sigma(\text{ch} + \langle s \rangle), && \text{by definition of minimal state space} \\
&\Rightarrow \sigma(\text{Eq}(\text{ch}, s)) = f_Q(\sigma(\text{ch}), s), && \text{by definition of } \sigma \\
&\Rightarrow (\sigma(\text{Eq}(\text{ch}, s)), s) = (f_Q(\sigma(\text{ch}), s), s), \\
&\Rightarrow \sigma'(\text{Eq}'(\text{ch}, s), s) = f_Q'(\sigma'(\text{ch}, s)), && \text{by definition of } \text{Eq}', f_Q', \text{ and } \sigma'.
\end{aligned}$$

Also, $\sigma'(\phi, \epsilon) = (\sigma(\phi), \epsilon) = (q_1, \epsilon)$ Thus, σ' is a homomorphism from $\mathbb{E}_b$ to $\mathbb{T}_s$. $\square$

Theorem (ET-Isomorphism) If $\beta(f_b, f_s)$, where f_b has a unique canonicalization (CH, Cn) and $\Delta f_s.Q = \sigma(\text{CH})$, then function $\sigma': \text{CH} \times S \rightarrow Q \times S$ defined by $\sigma'(\text{ch}, s) = (\sigma(\text{ch}, s), s)$ is an isomorphism from $\mathbb{E}_b$ to $\mathbb{T}_s$.

Proof Since (CH, Cn) is a unique canonicalization and $\Delta f_s.Q = \sigma(\text{CH})$, according to Minimized State Space Theorem and the ET-Homomorphism Theorem, σ' is a homomorphism from $\mathbb{E}_b$ to $\mathbb{T}_s$. Since $\Delta f_s.Q = \sigma(\text{CH})$, σ' is onto. Now, prove σ' is a one-to-one mapping by contradiction. Assume the assertion is false, i.e., (CH, Cn) is a unique canonicalization, but σ' is not a one-to-one.

$$\begin{aligned}
&\exists \text{ch}_1, \text{ch}_2 \in \text{CH} \exists s \in S [\text{ch}_1 \neq \text{ch}_2 \wedge \sigma'(\text{ch}_1, s) = \sigma'(\text{ch}_2, s)] \\
&\Rightarrow \exists \text{ch}_1, \text{ch}_2 \in \text{CH} \exists s \in S [\text{ch}_1 \neq \text{ch}_2 \wedge (\sigma(\text{ch}_1), s) = (\sigma(\text{ch}_2), s)], \\
&\quad \text{by the definition of unique canonicalization} \\
&\Rightarrow \exists \text{ch}_1, \text{ch}_2 \in \text{CH} [\text{ch}_1 \neq \text{ch}_2 \wedge \sigma(\text{ch}_1) = \sigma(\text{ch}_2)]
\end{aligned}$$

which is a contradiction to State Equivalence Theorem. $\square$

Theorem (TT-Isomorphism) If $\beta(f_b, f_{s1})$ and $\beta(f_b, f_{s2})$, where f_b has a unique canonicalization (Ch, Cn) and both f_{s1} and f_{s2} are classical state boxes with state spaces $Q_1 = \sigma_1(\text{CH})$ and $Q_2 = \sigma_2(\text{CH})$, respectively, then $\mathbb{T}_{s1}$ is isomorphic to $\mathbb{T}_{s2}$.

Proof Let $E_b = (CH \times S, Eq', (\phi, \epsilon))$, $T_{s1} = (Q_1 \times S, f_{Q1}', (q_{11}, \epsilon))$, and $T_{s2} = (Q_2 \times S, f_{Q2}', (q_{21}, \epsilon))$.

Since (CH, Cn) is a unique canonicalization, $Q_1 = \sigma_1(CH)$, $Q_2 = \sigma_2(CH)$, and both f_{s1} and f_{s2} are classical, by the ET-Isomorphism Theorem, E_b is isomorphic to T_{s1} and E_b is isomorphic to T_{s2} . Since isomorphism is an equivalence relation on algebras [46], T_{s1} is isomorphic to T_{s2} . □

The goal of state design is to achieve (optimal) state encapsulation to support state maintenance and response production while satisfying other design criteria. The difference between canonical histories in CH and states in Q is that a canonical history is still a stimulus history containing only essential stimuli, whereas a state in Q encapsulates equivalent stimulus histories into an optimal abstract data structure which may no longer be a valid stimulus history and which might serve as a partial solution for future responses.

The Minimized State Space, the ET-Homomorphism, and the ET-Isomorphism theorems are *state encapsulation theorems* which are constraints on state design and provide theoretical guidance for determining states as an aggregation of canonical stimulus histories.

If a black box has a unique canonicalization (CH, Cn) , according to the theorems, $Q = \sigma(CH)$ must be minimal. σ must be a one-to-one mapping rather than a many-to-one aggregation of canonical histories. Such a one-to-one mapping can be a permutation of stimuli, a projection of distinct components of stimuli, or an encoding of stimuli in canonical histories. And any canonical history is recoverable from its corresponding state by $ch = \sigma^{-1}(q)$.

If a black box has a multiple canonicalization (CH, Cn) , any one-to-one mapping $\sigma: CH \rightarrow Q$ will result in a non-minimal state space. A unique canonicalization may be difficult to find or be undesirable if found for the black box largely due to the constraint that a canonical history must still be a valid stimulus history. In state design, there is no such constraint. Thus, a many-to-one σ that maps all equivalent canonical histories into a single

state is possible. If such a σ can be found, $Q=\sigma(CH)$ is minimal. This means that among equivalent canonical histories, the details of individual stimuli are not important, they all contain the same information. A state in Q is an optimal aggregation of equivalent canonical histories and a specific canonical history cannot be recovered from its corresponding state since the inverse function is not defined. For example, black box SumOfSquareSinceLastReset in Appendix A has a multiple canonicalization, but its state box has an optimal state space and σ maps all equivalent canonical histories to the same state.

4.3 State migration

State migration refers to a design decision made in defining a state box that various components of state data or subsets of state space will be aggregated in various ways and each aggregation will be hidden from and unavailable to specifications regarding all other aggregations. Conceptually, a classical state box with total state space would have been discovered first, and then its parts aggregated in various ways, however, there is no guarantee that the first step occurs.

If state migration occurs during the clear box design then the design decision is made relative to the classical state box inherited in the expansion, as will be detailed in the next chapter.

If a specific manner of state migration is specified by the state box, then the generalized state box provides that stimulus histories will be retained for use in the definition of the internal black box. Consequently, the clear box expansion of the state box will have to implement the state aggregations as specified.

The essence of a state box is to use a single state to encapsulate a stimulus history so that the internal black box machine becomes a function of a single stimulus (q, s) instead of a

stimulus history. The generalized state box was introduced to model state migration, and is not an end in itself. Its machine function is eventually represented by lower level black boxes each of which has a classical state box that encapsulates migrated states.

Let $f_{sb}:(Q \times S)^+ \rightarrow (Q \times R)$, where Q is a retained state space, be the machine function of a generalized state box f_s after states are migrated to $f_{1b}, f_{2b}, \dots, f_{jb}$, where $\{f_{1b}, f_{2b}, \dots, f_{jb}\} \subseteq \lambda(\{f_{sb}\})$. Each of these lower level black boxes has a classical state box with a state space Q_i for $i \in \{1..j\}$. The machine functions of these classical state boxes are:

$$\{ f_{1sb}:Q_1 \times (Q \times S) \rightarrow Q_1 \times R_1, f_{2sb}:Q_2 \times (Q \times S) \rightarrow Q_2 \times R_2, \dots, f_{jsb}:Q_j \times (Q \times S) \rightarrow Q_j \times R_j \},$$

and f_{sb} can be represented by these classical state boxes as follows for suitable functions g, g' :

$$\begin{aligned} f_{sb}(qsh) &= g(q, s, f_{1b}(qsh), f_{2b}(qsh), \dots, f_{jb}(qsh)) \\ &= g(q, s, f_{1sb}(q_1, (q, s)), f_{2sb}(q_2, (q, s)), \dots, f_{jsb}(q_j, (q, s))) \\ &= g'(q, q_1, q_2, \dots, q_j, s) \end{aligned}$$

where $qsh \in (Q \times S)^+$, (q, s) is the last stimulus in qsh , $q_1 \in Q_1, q_2 \in Q_2, \dots, q_j \in Q_j$, and g' is the machine function of a classical state box.

If in a generalized state box, all states are migrated to lower level black boxes, then $\Delta f_s.QL = \emptyset$, $\Delta f_s.Q = \{\epsilon\}$, a *null state*, and,

$$\begin{aligned} f_{sb}(qsh) &= g(\epsilon, s, f_{1b}(qsh), f_{2b}(qsh), \dots, f_{jb}(qsh)) \\ &= g'(s, f_{1b}(sh), f_{2b}(sh), \dots, f_{jb}(sh)). \end{aligned}$$

After compression of null components, $f_{sb}:(Q \times S)^+ \rightarrow Q \times R = f_{sb}:(\{\epsilon\} \times S)^+ \rightarrow (\{\epsilon\} \times R) = f_{sb}:S^+ \rightarrow R$.

In such a case, $\Delta f_{sb} = (\phi + SL, \phi + RL, \delta_{sb}, \chi_{sb}) = (SL, RL, \delta_b, \chi_{sb})$. Based on the definition of WFSB, we have,

$$\begin{aligned} f_b = f_s &= \{ (sh, r) \mid sh = \langle s_1, \dots, s_n \rangle \wedge sh \in SH \wedge r \in R \wedge \\ &\quad \exists qsh \in (Q \times S)^n [\pi_s(qsh) = sh \wedge \forall i \in \{2..n\} [\pi_Q(qsh_i) = f_Q(qsh_{1..i-1})] \wedge r = f_R(qsh)] \} \\ &= \{ (sh, r) \mid sh \in SH \wedge r \in R \wedge r = f_R(sh) \} = \{ (sh, r) \mid sh \in SH \wedge r \in R \wedge \chi_{sb}(sh, r) \} \end{aligned}$$

Thus, $\chi_{sb} \Leftrightarrow \chi_b$. Whether χ_{sb} is identical to χ_b depends on the original specification structure of χ_b and the state migration consideration in Δf_{sb} . If χ_b uses some lower level black boxes which happen to be the lower level black boxes that χ_{sb} wants to use to migrate states, then χ_{sb} and χ_b can be identical. In example Min_Max_Avg12 in Appendix A, the state box Min_Max_Avg12 migrated all the states to the lower level black box MinMaxAvg12 which is also used in the black box Min_Max_Avg12. Thus, χ_{sb} and χ_b are identical, and Δf_{sb} and Δf_b are identical.

On the other hand, if χ_{sb} uses different lower level black boxes from the ones that χ_b uses to specify the state machine function and migrate states, then χ_{sb} is a different representation of the same function from χ_b and thus χ_{sb} and χ_b are not identical but equivalent.

Theorem (*Conservation of State*) If $\beta(f_b, f_{s1})$ and $\beta(f_b, f_{s2}), f_b$ has a unique canonicalization,

$\Delta f_{s1}.Q$ is optimal, $\Delta f_{s2}.Q$ is retained, and there are $\{f_1, f_2, \dots, f_j\} \subset \lambda^*(\{f_{s2}\})$ where for $i \in \{1..j\}$, $\Delta f_{is}.Q$, denoted Q_i , is an encapsulation of $\Delta f_b.SH$, then there exists a one-to-one mapping from $\Delta f_{s1}.Q$ to $(\Delta f_{s2}.Q) \times Q_1 \times Q_2 \times \dots \times Q_j$.

Proof Let $Q_s = \Delta f_{s2}.Q$. Consider two cases to reduce the generalized state box f_{s2} to a classical state box f_{s2}' . In the first case, let all black boxes $\{f_1, f_2, \dots, f_j\}$ be directly used by f_{s2} . Based on the definition of WFSB, there is a classical state box for each of these lower level black boxes and the machine functions of these classical state boxes can be described by:

$$\{ f_{1sb}: Q_1 \times (Q_s \times S) \rightarrow Q_1 \times R_1, f_{2sb}: Q_2 \times (Q_s \times S) \rightarrow Q_2 \times R_2, \dots, f_{jsb}: Q_j \times (Q_s \times S) \rightarrow Q_j \times R_j \},$$

and then, the machine function f_{sb} of f_{s2} can be represented by these classical state boxes as follows for some function g :

$$f_{sb}(qsh) = g(q, s, f_{1b}(qsh), f_{2b}(qsh), \dots, f_{jb}(qsh))$$

$$= g(q, s, f_{1sb}(q_1, (q, s)), f_{2sb}(q_2, (q, s)), \dots, f_{jsb}(q_j, (q, s))),$$

where $qsh \in (Q_S \times S)^+$, (q, s) is the last stimulus in qsh , and $q_1 \in Q_1, q_2 \in Q_2, \dots, q_j \in Q_j$. In the second case, let $f_{s2}, f_1, f_2, \dots, f_j$ form a single-path specification hierarchy. Based on the definition of WFSB, each of lower level black boxes $\{f_1, f_2, \dots, f_{j-1}\}$ has a generalized state box and f_j has a classical state box. The machine functions of these lower level state boxes can be described by: $\{f_{1sb} : (Q_1 \times Q_S \times S)^+ \rightarrow Q_1 \times R_1,$

$$f_{2sb} : (Q_2 \times Q_1 \times Q_S \times S)^+ \rightarrow Q_2 \times R_2, \dots, f_{jsb} : Q_j \times Q_{j-1} \times \dots \times Q_1 \times Q_S \times S \rightarrow Q_j \times R_j \},$$

and then, the machine function f_{sb} of f_{s2} can be represented by these lower level state boxes as follows for some function g :

$$f_{sb}(qsh) = g(q, s, f_{1b}(qsh))$$

$$= g(q, s, f_{1sb}(q_1, q, s, f_{2b}(qsh_1)))$$

$$= g(q, s, f_{1sb}(q_1, q, s, f_{2sb}(q_2, q_1, q, s, f_{3b}(qsh_2))))$$

$$= \dots$$

$$= g(q, s, f_{1sb}(q_1, q, s, f_{2sb}(q_2, q_1, q, s, \dots, f_{jsb}(q_j, q_{j-1}, \dots, q_1, q, s))..))$$

where $qsh \in (Q_S \times S)^+$, (q, s) is the last stimulus in qsh , $q_1 \in Q_1, q_2 \in Q_2, \dots, q_j \in Q_j$, and for $i \in \{1..j-1\} [qsh_i \in (Q_i \times Q_{i-1} \times \dots \times Q_1 \times Q_S \times S)^+]$.

In either case, $f_{sb}(qsh)$ has been reduced to $f_{sb}(qsh) = g'(q, q_1, q_2, \dots, q_j, s)$, a function on $Q_S \times Q_1 \times Q_2 \times \dots \times Q_j \times S$, hence a machine function of f_{s2}' . Any other cases of specification hierarchy of f_{s2} can be reduced in a similar way. Let $Q' = \Delta f_{s2}' \cdot \sigma(CH) \subseteq Q_S \times Q_1 \times Q_2 \times \dots \times Q_j$.

According to the Minimized State Space Theorem, Q' is minimal. Since both f_{s1} and f_{s2}' are classical state boxes, based on the TT-Isomorphism Theorem, there is a one-to-one mapping from $\Delta f_{s1} \cdot Q$ to Q' . Thus, there exists a one-to-one mapping from $\Delta f_{s1} \cdot Q$ to $(\Delta f_{s2} \cdot Q) \times Q_1 \times Q_2 \times \dots \times Q_j$. □

The Conservation Theorem addresses both subset-migration and component-migration.

In the case of subset-migration in the above theorem, if $\Delta f_{s1}.Q - \Delta f_{s2}.Q$ is partitioned into state spaces of lower level state boxes $f_{1s}, f_{2s}, \dots, f_{js}$, there will be a one-to-one mapping from

$(\Delta f_{s2}.Q) \times Q_1 \times Q_2 \times \dots \times Q_j$ to $(\Delta f_{s2}.Q) \cup Q_1 \cup Q_2 \cup \dots \cup Q_j$ by mapping $(q_{0,i}, \epsilon, \dots, \epsilon)$ to $q_{0,i}$,

$(\epsilon, q_{1,i}, \epsilon, \dots, \epsilon)$ to $q_{1,i}$, ..., and $(\epsilon, \dots, \epsilon, q_{j,i})$ to $q_{j,i}$, where $q_{0,i} \in \Delta f_{s2}.Q$ and $q_{k,i} \in Q_k$ for some i

and for $1 \leq k \leq j$. Hence, there is a one-to-one mapping from $\Delta f_{s1}.Q$ to $(\Delta f_{s2}.Q) \cup Q_1 \cup Q_2 \cup \dots \cup Q_j$.

The Conservation Theorem indicates that no matter how lower level state boxes encapsulate the migrated state, any state subsets or state components of the original classical state box should have its image in some of the lower level state boxes under a one-to-one mapping after the migration.

4.4 State Box Design

Given a black box definition $\Delta f_b = (SL, RL, \delta, \chi)$, a state box $\Delta f_s = (\Delta f_b, QL, q_1, \Delta f_{sb})$ can be defined by box structure expansion as follows. Box structure expansion will determine some aspects of the state box, the characteristics of the black box will determine other aspects, and the still other aspects must be the result of new invention. To some extent, the foregoing theorems provide insight and advice that will facilitate state box design.

4.4.1 Domain, Range and Interface

As described in Chapter 3, a black box and its state box have exactly the same external view: same function and same application interface. Therefore, the stimulus variable list SL , the response variable list RL and the domain predicate δ of the state box are simply inherited from the black box being expanded.

4.4.2 State Space

In order to define the state data, first the dependency of response on the prior stimulus history must be analyzed. Usually, a response is determined by a subhistory rather than the whole stimulus history. The state is then based on the encapsulation of critical subhistories.

If the black box is a CH-based description, the response dependency is clear. Based on the theorems in chapter 3, the canonical form of a prior stimulus history is necessary and sufficient for generating a response. Thus, the canonical form (CH, Cn) is adequate for state encapsulation. For the CH-based black box Stack in Appendix A, $\langle (PUSH, x_1), (PUSH, x_2), \dots \rangle$ is defined as the canonical stimulus history, which can be encapsulated as the state.

If the black box is an SH-based description, the response dependency must be determined from the black box mapping predicate. For example, in the SH-based specification of BirthdayBook, in each case of the mapping predicate, the response depends on list *Known* which is a sublist of the prior stimulus history *h*. After some analysis, one concludes that list *Known* should be encapsulated as state.

The state space so discovered must be *total* so that the internal black box can be defined. If it is not total then f_{sb} cannot be defined so that the $f_b = f_s$ requirement is met. If it is minimal, further design steps will be made from the best possible base; however, it is not disastrous for the state space to be larger than necessary.

If the implementation of the state box (i.e., the clear boxes to be expanded from it) is to be constrained with respect to information hiding, then certain (perhaps all) of the state space must be migrated.

State box design discovers the information that must be captured from the stimulus histories and then specifies the state space design in (QL, q_1) . Whether this state space specifies state migration or not will greatly influence the final step in state box design. In any

case, further details of the structure and representation of the data are deferred until clear box design.

4.4.3 Internal Black Box

The state box definition $\Delta f_s = (\Delta f_b, (QL, q_1), \Delta f_{sb})$ will be complete with the definition of $\Delta f_{sb} = (QL + \Delta f_b.SL, QL + \Delta f_b.RL, \delta_{sb}, \chi_{sb})$ so that $f_s = f_b$. Theorems and specification techniques introduced in chapter 3 for black box specification are applicable to the internal black box of a state box.

Although δ_{sb} and χ_{sb} can be written in a totally different specification structure from the δ_b and χ_b as long as they define the same external function, BDM recommends that if practical, δ_{sb} and χ_{sb} be transformed from δ_b and χ_b so that they have the same specification structure (domain partitioning and specification hierarchy) in the form of normal black boxes. Such a similarity between the two specification structures will make both the design and the verification of a state box easier.

At this point, a state box definition becomes a state-based specification with three parts $(Q \times S)$, $f_Q: (Q \times S)^+ \rightarrow Q$, and $f_R: (Q \times S)^+ \rightarrow R$. There are three general cases; all state is migrated, none is migrated, and some but not all is migrated.

As shown above, $(QL, q_1) = (\phi, \epsilon)$ if and only if all the state space is to be migrated.

If there is no state migration, (QL, q_1) is total and only the current state and current stimulus $(Q \times S)$ are needed. δ_b and χ_b are expressed in terms of operations on S^+ . δ_{sb} and χ_{sb} are expressed in terms of operations on $Q \times S$. In passing from δ_b to δ_{sb} and from χ_b to χ_{sb} , first the correspondence between the operations on S^+ and operations on $Q \times S$ are established. Then, stimulus histories and operations on stimulus histories in δ_b and χ_b are replaced by the

states and the corresponding operations on states to produce a $Q \times S$ based description of δ_{sb} and χ_{sb} .

δ_{sb} is defined so that $\forall h+\langle s \rangle \in SH \forall q \in Q [q = \sigma(h) \Rightarrow (\delta_{sb}(q, s) \Leftrightarrow \delta_b(h+\langle s \rangle))]$.

χ_{sb} can be defined by $\chi_Q \wedge \chi_R$ for f_{sb} based on f_Q and f_R . State mapping predicate $\chi_Q: (Q \times S) \times Q \rightarrow \{\text{TRUE}, \text{FALSE}\}$ defines final states for the state variables. According to the state encapsulation concepts $\sigma(\text{Eq}(ch, s)) \equiv f_Q(\sigma(ch), s)$. Thus, f_Q corresponds to Eq. If the black box is a CH-based description, Eq is readily available. If the black box is a SH-based description, a CH and Eq can be defined after an analysis of the equivalence relation among stimulus histories in the SH-based description. Sometimes, it is not necessary to construct a CH and Eq first. Since Q is an encapsulation of some critical sublists of the stimulus history, χ_Q can be directly invented based on the semantics of Q and the relationship among the sublists and the whole stimulus history defined in the SH-based description.

Response mapping predicate $\chi_R: (Q \times S) \times R \rightarrow \{\text{TRUE}, \text{FALSE}\}$ defines final states for the response variables in terms of a current state and a current stimulus. According to the definition of WFSB, $\forall h+\langle s \rangle \in SH [f_b(h+\langle s \rangle) = f_R(\sigma(h), s)]$. Thus, for any $h+\langle s \rangle \in SH$, any $r \in R$, and $q = \sigma(h)$, $\chi_R(q, s, r) = \chi_R(\sigma(h), s, r)$ can be derived from $\chi_b(h+\langle s \rangle, r)$. In this way, history-based mapping χ_b is transformed to state-based mapping χ_R and the normal black box specification structure remains unchanged.

If there is some state migration, then invention of the state box proceeds as follows.

δ_{sb} is defined such that

$$\forall qsh \in (Q \times S)^+ [\delta_{sb}(qsh) \Leftrightarrow (\delta_b(\pi_s(qsh)) \wedge \forall i \in \{2.. \#qsh\} [\pi_Q(qsh_i) = f_Q(qsh_{1..i-1})])].$$

where the subformula $\forall i \in \{1..n-1\} [q_{i+1} = f_Q(qsh_{1..i})]$ is guaranteed to be true since the state histories are produced by f_{sb} itself.

State mapping predicate $\chi_Q: (Q \times S) \times Q \rightarrow \{\text{TRUE}, \text{FALSE}\}$ defines a mapping from the current state and current stimulus to a new state. The prior state-stimulus history is not needed here since the current state already contains all the information in the history needed for producing a new (partial) state. Since Q is just a partial state space defined after some state components are migrated to lower level black boxes from a total state space, χ_Q can be invented in the same way as that of a classical state box, except it is a slice of the state mapping predicate of the classical state box.

Response mapping predicate $\chi_R: (Q \times S)^+ \times R \rightarrow \{\text{TRUE}, \text{FALSE}\}$ defines final states for the response variables in terms of state-stimulus history. Let $f_{1b}, f_{2b}, \dots, f_{jb}$ be lower level black boxes that encapsulate migrated state components. Generally, for $s \in S$, $r \in R$, $q \in Q$, and $qsh \in (Q \times S)^+$, χ_{sb} can be defined as $r = g(q, s, f_{1b}(qsh), f_{2b}(qsh), \dots, f_{jb}(qsh))$ for some function g such that $\forall sh \in \text{dom}(f_b) \forall qsh \in \text{dom}(f_{sb}) [sh = \pi_s(qsh) \Rightarrow f_b(sh) = f_R(qsh)]$. There are two possible situations. If $f_{1b}, f_{2b}, \dots, f_{jb}$ are used in χ_b as functions of stimulus history $sh \in SH$, they can be transformed to functions of $qsh \in (Q \times S)^+$, and χ_R can be transformed from χ_b by replacing stimulus histories and operations on stimulus histories in χ_b by the states and the state operations, except all application interfaces to $f_{1b}, f_{2b}, \dots, f_{jb}$, where arguments sh are replaced by qsh . If $f_{1b}, f_{2b}, \dots, f_{jb}$ are not used in χ_b , then the specification structure of χ_{sb} is different from that of χ_b . In such a case, there is no general transformation from χ_b to χ_{sb} and χ_{sb} is just a black box specification constrained by χ_b and Q to be defined using BDM black box specification methods.

Combine χ_Q and χ_R . χ_Q and χ_R can be combined to form χ_{sb} in one of two ways. If the specification structures of χ_Q and χ_R are similar, χ_Q and χ_R can be merged and intertwined to form χ_{sb} . Sometimes, even though the specification structures of χ_Q and χ_R are different, one of them can be transformed to make their structures similar if there is a benefit to having

them merged. In any case, χ_{sb} can be defined by connecting χ_Q and χ_R using logical conjunction such that $\chi_{sb} = \chi_Q \wedge \chi_R$.

4.5 State Box Verification

State box verification is to verify the correctness of the state box with respect to the black box from which it was expanded. The preciseness of the syntax and the semantics of the well formed black boxes and well formed state boxes in BDM facilitates such verification in greater formality.

The state box verification is focused on verifying the correctness of a state box f_s with respect to its black box f_b , i.e., to prove that $f_b = f_s$. If $f_b = f_s$, then the completeness and consistency of f_s are implied by the completeness and consistency of f_b . Moreover, the Completeness and Consistency Theorem, Normal Partition Theorem, and Normal BB Verification Theorems introduced in Chapter 3 are applicable to the internal black box f_{sb} of the state box f_s .

Since f_b and f_s are defined by the characteristic functions χ_b and χ_{sb} , $f_b = f_s$ is established by proving that

$$\forall (h+\langle s \rangle) \in SH \forall r \in R [\chi_b(h+\langle s \rangle, r) \Leftrightarrow \chi_{sb}((\sigma(h), s), (\sigma(h+\langle s \rangle), r))].$$

Mills proposed a general approach of verifying a state box by first deriving a black box from the state box through eliminating the state data, and then comparing the derived black box to the original black box. The current BSM verification methods in the literature follow Mills' approach. Since $\sigma: SH \rightarrow Q$ is a many-to-one mapping, to eliminate state data Q one must use $\sigma^{-1}: Q \rightarrow \mathcal{P}(SH)$ which is not easy to define. Indeed, when state migration is specified the verification task becomes futile. Furthermore, such a derive-then-compare

approach introduces one more box, the derived black box, for every state box and thus creates another artifact to track in the design record.

In BDM, a different approach to state box verification is proposed. The proof is conducted directly on the original black box and the state box to be verified using the following algorithm.

State Box Verification Method

Given $\beta(f_b, f_s)$, f_s can be verified against f_b as follows. For each subdomain SH_i of the domain partitioning defined in the black box mapping predicate χ_b , perform:

- A. *For an arbitrary stimulus history $h+\langle s \rangle$ in the subdomain, derive a response r from χ_b and $h+\langle s \rangle$ such that $\chi_b(h+\langle s \rangle, r)=TRUE$.*
- B. *Map h to a state q using the encapsulation function σ based on χ_Q , and derive a response r' from χ_R and (q, s) such that $\chi_R(q, s, r')=TRUE$.*
- C. *Compare r and r' . If $r \neq r'$, then $f_b \neq f_s$.*

If $r=r'$ for all subdomains, then $f_b=f_s$.

Since the number of subdomains in a domain partitioning of a black box specification is usually small, the number of such proofs is usually acceptable.

Since the specification structures of black box BirthdayBook and its state box defined previously are similar, the verification of the state box is straight forward. However, the specification structures of black box Stack and its state box are different. Thus, the verification of state box Stack is more difficult. The state box verification algorithm is illustrated in the following example.

Example Verify state box Stack. The χ_b has two subdomains SH_1 and SH_2 partitioned by the condition formulas of the root case predicate.

Subdomain SH₁ :

Step A: for any stimulus history $h+\langle s \rangle$ in SH_1 , we have

$$\begin{aligned} & \chi_b \wedge h+\langle s \rangle \in SH_1 \\ \Rightarrow & \chi_b \wedge s.Command \in \{POP, TOP\} \wedge i \neq \epsilon \\ \Rightarrow & return = h_i.Value \end{aligned}$$

Step B: $\sigma(h) = stack^*$

$$\begin{aligned} = & (stack' + \langle h_i.Value \rangle \mid stack' = \sigma(h_{1..i-1}) \wedge \\ & i = \text{Max}(\{ j \mid h_j = (PUSH, *) \wedge (CREATE) \notin h_{j+1..s} \wedge \\ & \text{Count}(_ = (POP), h_{j+1..s}) = \text{Count}(_ = (PUSH, *), h_{j+1..s}) \})) \\ \Rightarrow & \#stack^* \neq 0, \text{ by } i \neq \epsilon. \end{aligned}$$

Thus, we have the derivation:

$$\begin{aligned} & \chi_R \wedge (s.Command = (POP) \vee s.Command = (TOP)) \wedge \#stack^* \neq 0 \\ \Rightarrow & return = stack^*_s \\ \Rightarrow & return = h_i.Value \end{aligned}$$

Step C: Comparing the responses derived from step A and step B,

$$\forall sh \in SH_1 [Stack_b(sh) = Stack_s(sh)].$$

Subdomain SH₂:

Step A: for any stimulus history $h+\langle s \rangle$ in SH_2 , we have

$$\begin{aligned} & \chi_b \wedge h+\langle s \rangle \in SH_2 \\ \Rightarrow & \chi_b \wedge \neg (s.Command \in \{POP, TOP\} \wedge i \neq \epsilon) \\ \Rightarrow & return = \epsilon \end{aligned}$$

Step B: Since

$$\begin{aligned} & \chi_b \wedge h+\langle s \rangle \in SH_2 \\ \Rightarrow & \neg (s.Command = (POP) \vee s.Command = (TOP)) \vee i = \epsilon \end{aligned}$$

$$\Leftrightarrow \neg(s.Command=(POP) \vee s.Command=(TOP)) \vee$$

$$(s.Command=(POP) \vee s.Command=(TOP)) \wedge i=\epsilon,$$

two cases need to be considered.

Case 1: $\neg(s.Command=(POP) \vee s.Command=(TOP))$, we have the derivation

$$\chi_R \wedge \neg(s.Command=(POP) \vee s.Command=(TOP))$$

$$\Rightarrow \chi_R \wedge (s.Command=(CREATE) \vee s.Command=(PUSH,*))$$

$$\Rightarrow \text{return}=\epsilon$$

Case 2: $(s.Command=(POP) \vee s.Command=(TOP)) \wedge i=\epsilon$, we have,

$$i=\epsilon$$

$$\Leftrightarrow \neg \exists i[h_i=(PUSH,*) \wedge (CREATE) \notin h_{i+1..s} \wedge$$

$$\text{Count}(_=(POP), h_{i+1..s}) = \text{Count}(_=(PUSH,*), h_{i+1..s})]$$

$$\Rightarrow \sigma(h)=\text{stack}^\circ=\phi$$

$$\Rightarrow \#stack^\circ=0.$$

Thus, we have the derivation

$$\chi_R \wedge (s.Command=(POP) \vee s.Command=(TOP)) \wedge \#stack^\circ=0$$

$$\Rightarrow \text{return} = \epsilon$$

Step C: Comparing the responses derived from step A and step B,

$$\forall sh \in SH_2[\text{Stack}_b(sh) = \text{Stack}_s(sh)].$$

Since $SH = SH_1 \cup SH_2$, this concludes the verification of the state box Stack_s with respect to the black box Stack_b such that $\forall sh \in SH[\text{Stack}_b(sh) = \text{Stack}_s(sh)]$.

In summary the state box verification in BDM has the following advantages:

1. The verification consists of formal proofs.

2. The correctness of a state box with respect to a black box is proved by checking one representative stimulus history from each of a finite number of equivalence classes defined by the domain partitioning, instead of contemplating an infinite number of possible stimulus histories.
3. The verification is performed directly on the black box and the state box to be verified. The derived black box is eliminated.

Chapter 5

BDM Clear Boxes

A clear box is an expansion of a state box. It has the identical external behavior as its state box. The clear box represents a procedure that decomposes the state box function to several black boxes at the next lower level of the usage hierarchy and defines a sequential or a concurrent usage of these lower level black boxes.

This chapter presents the semantics of well formed clear boxes. Four prime control structures *sequence*, *alternation*, *iteration*, and *concurrent*, are used as building blocks for procedural implementation during a clear box design as well as program parts for function abstraction of the clear box during functional verification. Usage hierarchy and related concepts are defined. The fundamental design principles of referential transparency, state migration, and common services are reviewed. A stepwise function abstraction and function composition method is proposed to assist functional verification of clear boxes.

5.1 Clear Box Semantics

Definition A *well formed clear box*, denoted f_c , has definition denoted Δf_c , and

$\Delta f_c = (\Delta f_s, (TL, t_1), P)$, where Δf_s is WFSB, $TL = (tT_1 T_1, tT_2 T_2, \dots, tT_k T_k)$ is a finite list of concrete state variable declarations, t_1 is the initial state of $T \subseteq t(T_1)_e \times \dots \times t(T_k)_e$, the set of all possible concrete states, P is a C++ program on the data space of Δf_s , and $f_c = f_s$.

Since a clear box is an expansion of a state box and the state box is, in turn, an expansion of a black box, the fully detailed definition of a clear box is:

$$((SL, RL, \delta_b, \chi_b), (QL, q_1), (QL+SL, QL+RL, \delta_{sb}, \chi_{sb}), (TL, t_1), P).$$

TL is an implementation of the state QL of the state box. A mapping $\tau: Q \rightarrow T$, is called an *implementation function*. If $TL=\phi$, T is defined as $\{\epsilon\}$. t_1 is the *initial state* of T and $t_1=\tau(q_1)$.

The clear box procedure P is an implementation of $\Delta f_s f_{sb}$. The function computed by P is denoted "[P]" following Linger [36]. In general, P can be made up of three program slices P_δ , P_R and P_T so that $P \Leftrightarrow \{ DL; \text{if } P_\delta \text{ then con } (P_{RT}) \}$. $DL=(tD_1 D_1, tD_2 D_2, \dots, tD_j D_j)$ is a finite list of local variable declarations, d_1 is the set of initial or default values of $D = tD_1 \times tD_2 \times \dots \times tD_j$, the set of all possible local variables, and $\text{con } (P_{RT})$ is a C++ representation of concurrent execution of program slices P_R and P_T . The phrase "P is a C++ program on the data space of Δf_s " in the definition of a WFCB means that P only references data in S, R, T and D.

$[P_\delta]:(T \times S)^+ \rightarrow \{\text{TRUE}, \text{FALSE}\}$ is a data validation program that defines the domain of [P] which is derived from $\Delta f_s \delta_{sb}$. The function $[P]:(T \times S)^+ \rightarrow (T \times R)$ is related to f_{sb} as follows:

$$\forall qsh \in \text{dom}(f_{sb}) \forall tsh \in \text{dom}([P]) [\pi_s(qsh) = \pi_s(tsh) \Rightarrow (\tau(\pi_Q f_{sb}(qsh)), \pi_R f_{sb}(qsh)) = [P](tsh)].$$

P_R is the response mapping program with function $[P_R]:(T \times S)^+ \rightarrow R$, and P_T is a state mapping program with function $[P_T]:(T \times S)^+ \rightarrow T$. They are related to the pair of functions (f_R, f_Q) of f_{sb} as follows:

$$\forall qsh \in \text{dom}(f_{sb}) \forall tsh \in \text{dom}([P]) [\pi_s(qsh) = \pi_s(tsh) \Rightarrow f_R(qsh) = [P_R](tsh)], \text{ and}$$

$$\forall qsh \in \text{dom}(f_{sb}) \forall tsh \in \text{dom}([P]) [\pi_s(qsh) = \pi_s(tsh) \Rightarrow \tau(f_Q(qsh)) = [P_T](tsh)].$$

Thus, the external behavior of the clear box function f_c can be recovered by a pair of functions $([P_R], [P_T])$ used to construct $[P]$:

$$f_c = \{ (sh, r) \mid sh = \langle s_1, \dots, s_n \rangle \wedge sh \in SH \wedge r \in R \wedge \\ \exists tsh \in (T \times S)^n [\pi_S(tsh) = sh \wedge \forall i \in \{2..n\} [\pi_T(tsh_i) = [P_T](tsh_{1..i-1})] \wedge r = [P_R](tsh)] \}$$

If P produces each response based on only the current stimulus and the internal state, P implements a classical state box, $[P]: T \times S \rightarrow T \times R$, such that

$$f_c = \{ (sh, r) \mid sh = \langle s_1, \dots, s_n \rangle \wedge sh \in SH \wedge r \in R \wedge \forall i \in \{1..n\} [t_{i+1} = [P_T](t_i, s_i)] \wedge r = [P_R](t_n, s_n) \}.$$

We require that $f_c = f_s$. $[P]$ is constrained by f_{sb} and by the environment which includes the compiler, operating system and associated utilities. If the function of P is "simple", P can be written in C++ statements without using additional lower level black boxes. In such a case, the box expansion on that path has reached an end. Otherwise, the box expansion extends to the next level of the usage hierarchy.

For a given state box, there are many ways to design a state data structure of concrete type and a procedure conforming to the above outline for the clear box. A state box and its clear box have the identical external behavior which means that they have the same application interfaces and the same sets of ordered pairs.

5.2 Usage Hierarchy

In BDM design, a regular black box will be expanded to a state box and the state box will be expanded to a clear box in the format described above. In a clear box design, the function is decomposed into a sequence use or a concurrent use of next level of black boxes.

Definition f_c is a *box expansion* of f_s , denoted $\beta(f_s, f_c)$, if Δf_s is used in Δf_c .

Definition A BDM *box* is a triple $f = (f_b, f_s, f_c)$ such that $\beta(f_b, f_s) \wedge \beta(f_s, f_c)$.

Definition g_b is a *box expansion* of f_c , denoted $\beta(f_c, g_b)$, if the interface of g_b is used in

$$\Delta f_c.P.$$

Definition Let B be the set of all BDM boxes. For $f \in B$

$$\mu^0(f) = \{ f \}$$

$$\mu^1(f) = \mu^0(f) \cup \{ x \mid x \in B \wedge \beta(\pi_3 f, \pi_1 x) \}$$

$$\mu^k(f) = \mu^{k-1}(f) \cup \{ x \mid \exists y \in \mu^{k-1}(f) \wedge \beta(\pi_3 y, \pi_1 x) \}$$

$$\mu^*(f) = \mu^k(f) \text{ if } \mu^k(f) = \mu^{k+1}(f).$$

$\mu(f)$ is the order-1 usage set of f , i.e., f and all the boxes it uses directly.

$\mu^k(f)$ is the order- k usage set of f with all boxes it uses for k levels. μ^* refers to the complete usage set, or *usage closure*, regardless of the number of levels. A *usage hierarchy* of box f is the tree representation of $\mu^*(f)$.

Definition For f a BDM box, $\mu^*(f)$ is a BDM system.

Definition BDM box f is a *common service box* if and only if $\#\{ x \mid f \in \mu^*(x) \} > 1$.

5.3 Prime Control Structures

Although, any C++ declaration statements and control statements are valid to use in P , BDM defines a set of prime control structures for P . Mills [1-3] proposed four prime control structures *sequence*, *alternation*, *iteration*, and *concurrent*, which can be used in a clear box procedure. Linger [36], shows that any proper program, i.e., a flow chart program that has a single-entry, a single-exit, and at least one path through each node from the entry to the exit, can be written in terms of sequence, alternation, and iteration prime control structures. Each of the prime control structures corresponds to a defined program function. These basic prime control structures are building blocks for procedural implementation during clear box

design and program parts for function abstraction of the clear box during functional verification.

5.3.1 Sequence Clear Box

Consider $\Delta f_c = (\Delta f_s, (TL, t_1), P)$, where $P = \{B_1; B_2\}$ is a sequence program. Each of B_1 and B_2 can be either a black box or a series of C++ statements which can be abstracted as a black box function.

Let $\Delta B_{1b} = (TL+SL, TL+UL, \delta_1, \chi_1)$, and

$$\Delta B_{2b} = (TL+UL, TL+RL, \delta_2, \chi_2).$$

Then $B_{1b}:(T \times S)^+ \rightarrow (T \times U)$, and

$$B_{2b}:(T \times U)^+ \rightarrow (T \times R).$$

The function computed by the clear box procedure is: $[P] = [B_1] \circ [B_2]$.

5.3.2 Alternation Clear Box

Consider $\Delta f_c = (\Delta f_s, (TL, t_1), P)$, where $P = \{\text{if } (C) \text{ then } B_1 \text{ else } B_2\}$ is an alternation program. Each of C , B_1 and B_2 can be either a black box or a block of C++ statements which can be abstracted as a black box function.

Let $\Delta C_b = (TL+SL, (BOOL \text{ Return}), \delta, \chi)$,

$$\Delta B_{1b} = (TL+SL, TL+RL, \delta_1, \chi_1), \text{ and}$$

$$\Delta B_{2b} = (TL+SL, TL+RL, \delta_2, \chi_2).$$

Then $C_b:(T \times S)^+ \rightarrow \{\text{TRUE}, \text{FALSE}\}$,

$$B_{1b}:(T \times S)^+ \rightarrow (T \times R), \text{ and}$$

$$B_{2b}:(T \times S)^+ \rightarrow (T \times R).$$

The function computed by the clear box procedure is:

$$[P] = \{ \text{tsh}, (t,r) \mid \text{tsh} \in (T \times S)^+ \wedge (t, r) \in (T \times R) \wedge$$

$$[C(\text{tsh}) \rightarrow (t, r) = [B_1](\text{tsh}) \parallel \neg C(\text{tsh}) \rightarrow (t, r) = [B_2](\text{tsh})] \}$$

Another form of alternation is the case program where $\Delta C_b = (TL+SL, (\text{t}(v) \vee), \delta, \chi)$,

$\{c_1, c_2, \dots, c_n\}$ is a set of terms and $B_1, B_2, \dots, B_n$ are lower level black boxes mapping $(T \times S)^+$ to $(T \times R)$.

$$P = \{ \text{switch}(C) \{ \text{case } c_1: B_1; \text{case } c_2: B_2; \dots; \text{case } c_n: B_n; \} \}, \text{ and}$$

$$[P] = \{ \text{tsh}, (t,r) \mid \text{tsh} \in (T \times S)^+ \wedge (t, r) \in (T \times R) \wedge$$

$$[C(\text{tsh}) = c_1 \rightarrow (t, r) = [B_1](\text{tsh})$$

$$\parallel C(\text{tsh}) = c_2 \rightarrow (t, r) = [B_2](\text{tsh})$$

$$\parallel \dots$$

$$\parallel C(\text{tsh}) = c_n \rightarrow (t, r) = [B_n](\text{tsh})] \}$$

5.3.3 Iteration Clear Box

Consider $\Delta f_c = (\Delta f_s, (TL, t_1), P)$, where $P = \{ \text{while } (C) \text{ do } \{B\} \}$ is an iteration program.

Each of C and B can be either a black box or a block of C++ statements which can be abstracted as a black box function.

Let $\Delta C_b = (TL+SL, (\text{BOOL Return}), \delta, \chi)$, and

$$\Delta B_b = (TL+SL, TL+RL, \delta, \chi).$$

Then $C_b: (T \times S)^+ \rightarrow \{\text{TRUE}, \text{FALSE}\}$, and

$$B_b: (T \times S)^+ \rightarrow (T \times R).$$

The clear box procedure computes:

$$[P] = \{ \text{tsh}, (t,r) \mid \text{tsh} \in (T \times S)^+ \wedge (t, r) \in (T \times R) \wedge$$

$$[C(\text{tsh}) \rightarrow (t, r) = [P] \circ [B](\text{tsh}) \parallel \neg C(\text{tsh}) \rightarrow (t, r) = (t^*, r^*)] \}$$

5.3.4 Concurrent Clear Box

Consider $\Delta f_c = (\Delta f_c, (TL, t_1), P)$, where $P = \{\text{con}(B)\}$ is a concurrent program adapted for C++ such that $B = (B_1, \dots, B_n)$ are executed concurrently. Each B_i can be either a black box or a block of C++ statements which can be abstracted as a black box function.

For $i \in \{1..n-1\}$, let $\Delta B_{bi} = (TL+SL, TL+RL, \delta_i, \chi_i)$ so that $B_{bi}: (T \times S)^+ \rightarrow (T \times R)$, and let $\Delta B_{nb} = (TL+RL+\dots+TL+RL, TL+RL, \delta_n, \chi_n)$ so that $B_{nb}: (T \times R)^{n-1} \rightarrow (T \times R)$. The clear box procedure computes:

$$[P] = \{ \text{tsh}, (t, r) \mid \text{tsh} \in (T \times S)^+ \wedge (t, r) \in (T \times R) \wedge \\ \exists t_1, \dots, t_{n-1} \in T \exists r_1, \dots, r_{n-1} \in R [(t_1, r_1) = [B_1](\text{tsh}) \wedge \dots \wedge (t_{n-1}, r_{n-1}) = [B_{n-1}](\text{tsh}) \\ \wedge (t, r) = [B_n](t_1, r_1, \dots, t_{n-1}, r_{n-1})] \}.$$

Although the concurrent structure is important, its details have not been illuminated in the literature. Here, too, explication of the concurrent structure is deferred for future work.

5.4 Clear Box Design

Given $\Delta f_s = (\Delta f_b, (QL, q_1), \Delta f_{sb})$, a clear box $\Delta f_c = (\Delta f_s, (TL, t_1), P)$ can be defined by box structure expansion as follows. The stimulus variable list SL and the response variable list RL are inherited from the state box being expanded.

5.4.1 Concrete State Space

The state space T of a clear box is the implementation of the state space Q of its state box. Q is designed for its specification clarity. T is designed for its ability to be effectively represented and manipulated in C++. The design of the concrete state space T is a human invention step and no general algorithm can be given that will satisfactorily construct an

effective T from Q , and $\tau: Q \rightarrow T$, for an arbitrary state box. Design of T depends on knowledge of data structures [48, 49], C++, and relevant design criteria and objectives.

5.4.2 Procedure Definition

The procedure design is an important invention step in a clear box definition. The goal of clear box design is functionally correct procedural implementation of the state box specification, including efficiency, extendibility, reusability and other design criteria.

Two issues are involved in procedure design, algorithm design and function decomposition. The analysis and the design of algorithms is an extensive subject, key to good clear box designs, but beyond the scope of this work. BSM is based on decomposition of functions to form a usage hierarchy. Such a decomposition is essential to stepwise refinement and the verification.

Many design principles and techniques have been developed over the past 40 years. BSM has focused on the four design principles of transaction closure (addressed in Chapter 3), referential transparency, state migration, and common services. Many other principles are also worthy, e.g., minimum coupling and maximum cohesion. The following section addresses only the BSM design principles; however, other principles are conducive to formal treatment.

5.5 BSM Design Principles

5.5.1 Referential Transparency

Informal principle

Version 1 [2]: "In the delegation of any system part for design and implementation, all requirements should be specified explicitly and independently, so that no further communication or coordination is logically required to complete the system part."

Version 2 [50]: "Referential transparency occurs when a black box abstraction is completely defined within the clear box at the next higher level in the usage hierarchy. The black box is then logically independent of the rest of the system, and can be designed to satisfy a well-defined behavior specification."

In hardware design, an integrated circuit chip has a number of pins as an interface to the rest of the system. The chip can be replaced by another chip using the same pin-interface and having the same function while employing different internal design, which may result in improved quality as measured by energy consumption, reliability, and service life. These benefits of use, maintenance and development are consequences of the engineering design methods used in hardware.

The principles quoted above focus on the design process rather than the benefits they produce, although the benefits are discussed in context. If one maintains referential transparency during design, then one can reap the benefits of substitutability of equals for equals during use, and independence during development.

It is problematical to sharpen the informal definitions of others. However, the following is an effort to restate the principle precisely for BSM, but independently of BDM.

Referential transparency is a property of a black box relative to a clear box which uses it. The specification that is the black box does not appear in the expression of the clear box, only a reference to its use appears there. Let $(B;C)$ be a black box specification and a clear box that satisfies the specification. Let $(C; f(B_1,...,B_n))$ represent an implementation of C via the use of black boxes $\{B_1,...,B_n\}$. Let $(B_i;C_i)$ represent the specification B_i and an implementation C_i such that the function of C_i is identical to the function of B_i . B_i has referential transparency with respect to C if and only if (1) the function specification B_i is complete and consistent, (2) the usage interface for B_i is included in the specification, and (3) the correctness of C_i is necessary but not sufficient for the correctness of C .

All of this simply says that C has been defined in terms of the use of other functions, each of which is a correct implementation of its complete and consistent specification.

It is more effective to set a fixed outcome and then prescribe design constraints that will produce the outcome. In BDM, a related concept of referential transparency has been imposed on a black box, any state boxes in its beta-field, and clear boxes in the beta-fields of these state boxes, because all three forms for any BDM box have identical external behavior and identical user interfaces. A black box can be reimplemented (re-expanded) using new internal data structures and new algorithms. The old state box and clear box can be replaced by new ones without any changes to the rest of the system because the interfaces and the functions of the new boxes are identical to their old counterparts.

Formal principle

Referential transparency in BDM: If f is a BDM box, substitutability of equals for equals and independence in the BDM expansion of each black box in $\mu^*(f)$ will be

maintained if the following design discipline is observed: (1) f_b is complete and consistent, (2) all components of $\mu^*(f)$ are developed by BDM expansion and verification, and (3) every black box in $\mu^*(f)$ is complete and consistent.

5.5.2 State Migration

Informal principle

Version 1 [2]: "System data should be decentralized to the smallest system parts that do not require duplicating data updates. If, for geographic or security reasons, system data should be decentralized to smaller system parts, the system should be designed to ensure correctly duplicated updates."

Version 2 [50] "State data should be identified and stored in the system part (i.e., data abstraction) at the lowest level in the box structure hierarchy that includes all references to those data."

Again, the design advice is not tied to well defined, desired outcomes. Exactly what benefits in terms of development, use and maintenance is observance of this principle supposed to achieve? Since state data distribution is inextricably tied to function decomposition, it would seem that criteria of lowest system parts and avoidance of duplicative updates would lead to strange and undesirable function decomposition if observed in the absence of other design criteria. Also, various programming languages facilitate state migration in different ways and to different degrees. If there is no state migration at all, then one has a monolith of global data, which experience has shown to be an error prone design that is difficult to implement and maintain.

The benefits of state migration are (1) to make items of data inaccessible to units of code that should not access them, (2) to enforce a discipline of usage interfaces on the manner

in which units of code may access items of data, and (3) support functional decomposition by aggregating data in ways that make the sub-functions less complex than the composite.

BSM now permits state migration in the state box or clear box. As detailed in the previous chapter, state migration specified in the state box complicates specifications, and may frustrate clear box design as well. When state migration occurs in state box design, it is unlikely that balanced consideration of algorithmic and procedural issues will accompany consideration of data storage and access issues.

Formal principle

State Migration in BDM: In order to achieve effective function decomposition and information hiding, (1) β -expand and verify a classical state box first, (2) decide state migration in conjunction with function decomposition in the clear box, β -expand and verify, (3) replace the classical state box with a β -expanded state box that specifies state migration as planned in the clear box and verify to the black box, (4) β -expand the new state box and verify.

5.5.3 Common Services

Informal principle

Version 1 [2]: "System parts with multiple uses should be considered for definition as common services. A corollary principle is to create as many opportunities as possible for reusability within and between system parts".

Version 2 [50]: "A common service is a data abstraction that is described in a separate box structure hierarchy, and used in other box-structured systems. System parts with multiple uses should be defined as common services for reusability."

The benefits envisioned to result from this design principle are apparent: (1) Reuse of "properly designed" assets is more cost effective than new development, (2) function decomposition can be oriented toward existing system components, (3) newly developed system parts can be designed in general form to support reuse, and (4) greater uniformity in specification and use will result in greater reuse.

Two forms of common services can be developed, one based on function abstraction and another based on data abstraction. Function abstraction is commonly associated with mathematical computations. Data abstraction concerns abstract data types and operations defined on them.

Formal principle

Common Services (Function Abstraction): If there are subprograms $SP_1, \dots, SP_n$ (not necessarily identical), each of which is used in one or more clear boxes, and $[SP_1] = \dots = [SP_n] = g$, then g should be defined as a common service WFBB, and each occurrence of $SP_1, \dots, SP_n$ should be replaced by a use of g .

Common Services (Data Abstraction): If there are subprograms $SP_1, \dots, SP_n$, each of which is used in one or more clear boxes on a common state space T , input space S and output space R such that $[SP_1] = g_1, [SP_2] = g_2, \dots, [SP_n] = g_n$, then a new common service box g to encapsulate T should be defined as:

```

SB  g(in NAT TransactionType, SL, out RL)
{
  state
    T t=ti;

  map
    [ TransactionType=1 → (t, r) = g1(t°, s)
    || TransactionType=2 → (t, r) = g2(t°, s)
    ..
    || TransactionType=n → (t, r) = gn(t°, s)
  ]
}

```

and each occurrence of SP_i should be replaced by a use of $g(i, s)$, for $1 \leq i \leq n$.

5.6 Clear Box Verification

Clear box verification follows the methods of Linger [36], with minor adaptations to BDM and design techniques.

In general, the mathematical problem posed by verification is difficult to impossible; but, in particular, when development work is done in anticipation of verification it will range from easy to difficult. In all cases, verification is the best way to gain confidence that a clear box design is correct.

Various verification methods have been proposed ranging from formal mathematical proofs, such as Hoare's axiomatic method [43], to informal human inspection, such as code walkthrough [51]. Generally, the more formal the verification method used, the more time and effort for verification is required.

In order to balance formal procedure and economy of effort, Mills has proposed a verification method called *functional verification* [28, 36]. The method is "based on functional semantics and on the reduction of software verification to ordinary mathematical reasoning about sets and function as directly as possible" [28]. The functional verification method is used in BDM clear box verification.

5.6.1 Function Abstraction of Clear Boxes

Verification of a clear box to its state box is aggravated by the fact that the two functions are defined in two different kinds of languages. In BDM, a state box is a non-procedural description written in BDM language, while its clear box is a procedural implementation written in C++. As in BSM, functional verification of a clear box consists of first deriving a state box from the clear box and then comparing the derived state box to the intended state box. The derivation of a state box from a clear box is the process of reverse engineering. Forward engineering from a high-level representation to a low-level representation is relatively easier than reverse engineering from a low-level representation to a high-level representation.

BDM proposes a stepwise function abstraction and function composition (FAFC) technique based on BDM language to abstract function from clear box procedures.

FAFC method

Procedure P of a clear box f_c contains a sequence of subprograms $\langle P_1, P_2, \dots, P_n \rangle$. In order to abstract a program function $[P]$ from P , the following steps will be applied to each subprogram P_i in order starting from P_1 . Before execution of P_1 , $\forall v \in SL \cup RL \cup TL \cup DL [v = v^*]$, where v^* is the value of v before the execution of P .

Abstraction: For each subprogram P_i , abstract its program function $[P_i]$ by giving a characteristic function χ_i for $[P_i]$:

- $\forall v \in \text{SLORLUTLODL}$, use v' to represent its initial state before the execution of P_i ;
use v to represent its final state after the execution of P_i ,
- $\forall v \in \text{SLORLUTLODL}$, whose value is not changed during the execution of P_i ,
define $v=v'$.

Composition: The composite function of $[P_i]$ and $[P_{i+1}]$ is obtained by:

- If neither of $[P_i]$ or $[P_{i+1}]$ is described by a case predicate, then every occurrence of each $'''$ -decorated variable in $[P_{i+1}]$ is simply replaced by the final state of the same variable in $[P_i]$.
- If one or both of $[P_i]$ and $[P_{i+1}]$ are described by case predicates, then both χ_i and χ_{i+1} are treated as case predicates, since for a non-case-predicate formula A , $A \Leftrightarrow [\text{TRUE} \rightarrow A]$. Let $[P_i]$ be defined by $[C_1 \rightarrow A_1 \parallel \dots \parallel C_m \rightarrow A_m]$ and $[P_{i+1}]$ be defined by $[D_1 \rightarrow B_1 \parallel \dots \parallel D_n \rightarrow B_n]$, then the composition of $[P_i]$ and $[P_{i+1}]$ is defined by

$$[C_1 \wedge D_{11} \rightarrow B_{11} \parallel C_1 \wedge D_{12} \rightarrow B_{12} \parallel \dots \parallel C_1 \wedge D_{1n} \rightarrow B_{1n}$$

$$\dots$$

$$\parallel C_m \wedge D_{m1} \rightarrow B_{m1} \parallel C_m \wedge D_{m2} \rightarrow B_{m2} \parallel \dots \parallel C_m \wedge D_{mn} \rightarrow B_{mn}]$$

where, D_{ij} is D_j after every occurrence of each $'''$ -decorated variable in D_j is replaced by the final state of the same variable in A_i , and B_{ij} is B_j after every occurrence of each $'''$ -decorated variable in B_j is replaced by the final state of the same variable in A_i .

If each P_i is a statement, then such an FAFC process is a statement-based function abstraction. In such cases, it is simpler than the statement-based trace table method. If each P_i

is a block of statements, then such an FAFC is a block based process after abstraction of $[P_i]$ using the direct assertion method.

Examples

$$1) \quad P = \{ t=x; x=y; y=t; \}$$

P_i	Abstraction _i	Composition _i
$\{t=x;\}$	$[t=x' \wedge x=x' \wedge y=y']$	$[t=x^\circ \wedge x=x^\circ \wedge y=y^\circ]$
$\{x=y;\}$	$[x=y' \wedge t=t' \wedge y=y']$	$[x=y^\circ \wedge t=x^\circ \wedge y=y^\circ]$
$\{y=t;\}$	$[y=t' \wedge t=t' \wedge x=x']$	$[y=x^\circ \wedge t=x^\circ \wedge x=y^\circ]$

Thus, the program function of P is defined by: $BB [P](inout x, y, t) \{ y=x^\circ \wedge t=x^\circ \wedge x=y^\circ \}$.

2) Linger [36, pp243-248]] used the following program to demonstrate case-structured trace table.

$$P = \{ \text{if } (x \geq 0) \text{ then } x=x+y; \text{ else } y=y+x;$$

$$\text{if } (y \geq 0) \text{ then } y=x-y; \text{ else } x=y-x;$$

$$\text{if } (x \geq 0) \text{ then } x=x-y; \text{ else } y=y-x; \}$$

The same example to is used to illustrate the FAFC method. Each if-then-else statement is considered a subprogram P_i and the function abstraction and composition of the subprograms are illustrated below.

<u>Abstraction</u>	<u>Composition</u>
$[x' \geq 0 \rightarrow x=x'+y' \wedge y=y']$	$[x^\circ \geq 0 \rightarrow x=x^\circ+y^\circ \wedge y=y^\circ]$
$\parallel x' < 0 \rightarrow x=x' \wedge y=x'+y' \parallel$	$\parallel x^\circ < 0 \rightarrow x=x^\circ \wedge y=x^\circ+y^\circ \parallel$

$$[y' \geq 0 \rightarrow x = x' \wedge y = x' - y' \quad [x^\circ \geq 0 \wedge y^\circ \geq 0 \quad \rightarrow x = x^\circ + y^\circ \wedge y = x^\circ + y^\circ - y^\circ]$$

$$\parallel y' < 0 \rightarrow x = y' - x' \wedge y = y' \quad \parallel x^\circ < 0 \wedge x^\circ + y^\circ \geq 0 \quad \rightarrow x = x^\circ \wedge y = x^\circ - (y^\circ + x^\circ)$$

$$] \quad \parallel x^\circ \geq 0 \wedge y^\circ < 0 \quad \rightarrow x = y^\circ - (x^\circ + y^\circ) \wedge y = y^\circ$$

$$\parallel x^\circ < 0 \wedge x^\circ + y^\circ < 0 \quad \rightarrow x = y^\circ + x^\circ - x^\circ \wedge y = y^\circ + x^\circ]$$

$$\Leftrightarrow [x^\circ \geq 0 \wedge y^\circ \geq 0 \quad \rightarrow x = x^\circ + y^\circ \wedge y = x^\circ$$

$$\parallel x^\circ < 0 \wedge x^\circ + y^\circ \geq 0 \quad \rightarrow x = x^\circ \wedge y = -y^\circ$$

$$\parallel x^\circ \geq 0 \wedge y^\circ < 0 \quad \rightarrow x = -x^\circ \wedge y = y^\circ$$

$$\parallel x^\circ < 0 \wedge x^\circ + y^\circ < 0 \quad \rightarrow x = y^\circ \wedge y = y^\circ + x^\circ]$$

$$[x' \geq 0 \rightarrow x = x' - y' \wedge y = y' \quad [x^\circ \geq 0 \wedge y^\circ \geq 0 \wedge x^\circ + y^\circ \geq 0 \quad \rightarrow x = x^\circ + y^\circ - x^\circ \wedge y = x^\circ$$

$$\parallel x' < 0 \rightarrow x = x' \wedge y = y' - x' \quad \parallel x^\circ < 0 \wedge x^\circ + y^\circ \geq 0 \wedge x^\circ \geq 0 \quad \rightarrow x = x^\circ - (-y^\circ) \wedge y = -y^\circ$$

$$] \quad \parallel x^\circ \geq 0 \wedge y^\circ < 0 \wedge (-x^\circ) \geq 0 \quad \rightarrow x = -x^\circ - y^\circ \wedge y = y^\circ$$

$$\parallel x^\circ < 0 \wedge x^\circ + y^\circ < 0 \wedge y^\circ \geq 0 \quad \rightarrow x = y^\circ - (y^\circ + x^\circ) \wedge y = y^\circ + x^\circ$$

$$\parallel x^\circ \geq 0 \wedge y^\circ \geq 0 \wedge x^\circ + y^\circ < 0 \quad \rightarrow x = x^\circ + y^\circ \wedge y = x^\circ - (x^\circ + y^\circ)$$

$$\parallel x^\circ < 0 \wedge x^\circ + y^\circ \geq 0 \wedge x^\circ < 0 \quad \rightarrow x = x^\circ \wedge y = -y^\circ - x^\circ$$

$$\parallel x^\circ \geq 0 \wedge y^\circ < 0 \wedge (-x^\circ) < 0 \quad \rightarrow x = -x^\circ \wedge y = y^\circ - (-x^\circ)$$

$$\parallel x^\circ < 0 \wedge x^\circ + y^\circ < 0 \wedge y^\circ < 0 \quad \rightarrow x = y^\circ \wedge y = y^\circ + x^\circ - y^\circ]$$

$$\Leftrightarrow [x^\circ \geq 0 \wedge y^\circ \geq 0 \quad \rightarrow x = y^\circ \wedge y = x^\circ$$

$$\parallel x = 0 \wedge y^\circ < 0 \quad \rightarrow x = -x^\circ - y^\circ \wedge y = y^\circ$$

$$\parallel x^\circ < 0 \wedge x^\circ + y^\circ < 0 \wedge y^\circ \geq 0 \quad \rightarrow x = -x^\circ \wedge y = x^\circ + y^\circ$$

$$\parallel x^\circ < 0 \wedge x^\circ + y^\circ \geq 0 \quad \rightarrow x = x^\circ \wedge y = -x^\circ - y^\circ$$

$$\parallel x > 0 \wedge y^\circ < 0 \quad \rightarrow x = -x^\circ \wedge y = x^\circ + y^\circ$$

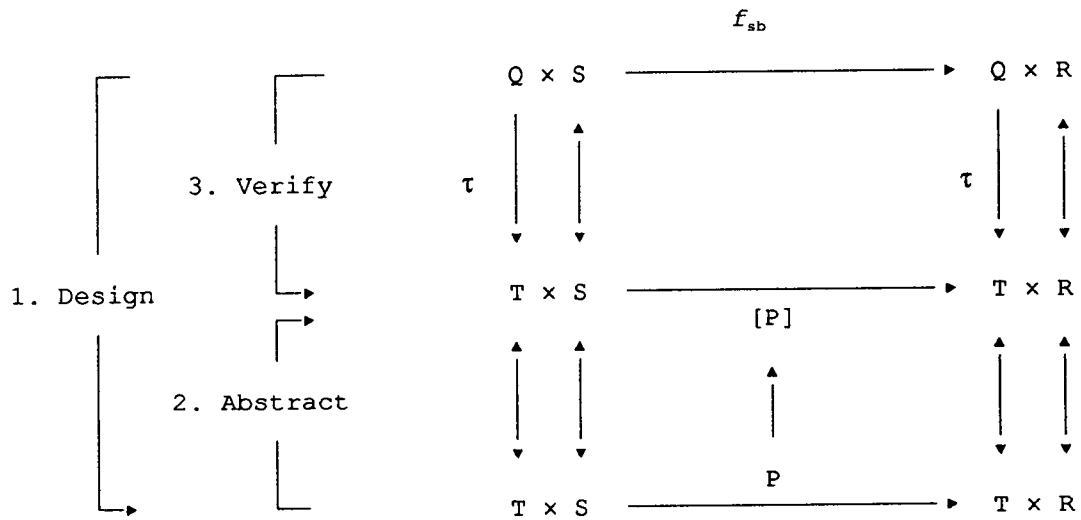
$$\parallel x^\circ < 0 \wedge y^\circ < 0 \quad \rightarrow x = y^\circ \wedge y = x^\circ]$$

Compared to the abstraction using the case-structured trace table method, where eight trace tables are used, one for each execution path, the abstraction using FAFC method seems somewhat simpler.

5.6.2 Function Verification of Clear Boxes

In BDM, verification of a clear box to its state box consists of first deriving a state box from the clear box and then comparing the derived state box to the intended state box. The comparison between the two state boxes is very similar to the comparison between a black box and its state box. The functions of both boxes are expressed in BDM language. The clear box verification is described by the following diagram and algorithm.

SB-CB Commuting Diagram



Clear Box Verification Method

Given a state box $\Delta f_s = (\Delta f_b, (QL, q_1), \Delta f_{sb})$ and a clear box $\Delta f_c = (\Delta f_s, (TL, t_1), P)$, the clear box f_c can be verified to f_s by the following algorithm:

A. Derive a state box $\Delta f'_s = (\Delta f'_b, (QL', q'_1), \Delta f'_{sb})$ from the clear box, where $\Delta f'_{sb} \cdot \chi'_{sb}$ is the mapping predicate abstracted from the clear box procedure P .

B. Compare the derived state box $\Delta f'_s$ to the source state box Δf_s by repeating the following steps. For each subdomain QS_i of the domain partitioning on $Q \times S$ defined in the mapping predicate χ_{sb} of the intended state box f_s , perform:

(1) For an arbitrary (q, s) in the subdomain, derive a response (nq, r) from χ_{sb} and

(q, s) such that $\chi_{sb}(q, s, nq, r) = \text{TRUE}$.

(2) Map q to a state q' in the state space Q' of the derived f'_s , and derive a response

(nq', r') from χ'_{sb} and (q', s) such that $\chi'_{sb}(q', s, nq', r') = \text{TRUE}$.

(3) Compare (nq, r) and (nq', r') . If $(\tau(nq), r) \neq (nq', r')$, then $f_s \neq f'_s$.

If $(\tau(nq), r) = (nq', r')$ for all subdomains, then $f_s = f'_s$.

Since the number of subdomains in a domain partitioning of a state box specification is usually small, the number of such proofs is usually acceptable.

Chapter 6

Conclusion

6.1 Synopsis

The goal of this research was to close the gap between the intuitively appealing concepts of the Box Structure Method (BSM) and the practical difficulties in developing software in a specific language for a specific environment by the available descriptions of BSM. On the one hand, every effort was made to remain faithful to the original statement of BSM; on the other hand, definitions were changed or replaced when necessary to facilitate the software development practice. This research presents an integrated formal method for BSM called the Box Development Method (BDM) where artifacts and techniques of BSM are represented in a form of predicate calculus and tailored to C++, the target implementation language. The contributions of BDM to software engineering are summarized as follows:

(1) Theoretical insights. Various mathematical properties of black boxes, state boxes and clear boxes and relationships among them have been established. The unified external views of three abstractions and the relationships between the external views and the internal views resolves many discontinuities in passing from concept to code. Use of case predicates, the analysis of equivalence relations on stimulus histories and the use of canonical forms of stimulus histories are tangible advice available to writers of black boxes. By directly relating the black box advice to state encapsulation, and the classical state box, writers of state boxes have been given tangible advice. Moreover, the considerable difficulties introduced by the generalized state box have been clarified, as have the technical issues related to state migration

specified in the state box versus state migration specified in the clear box. Conservation of state under state migration, and the relationship between abstract state and concrete state have resulted in specific advice for the clear box writer.

(2) Formal syntax and semantics. All parts of all three types of boxes have been specified in all details of syntax and semantics. This permits a seamless description from top level black box to lowest level lines of code with no syntactic or semantic doubt at any step. Many difficulties that arise in practice can be resolved through the use of partial functions, domain predicates, mapping predicates, null responses, and abstract and concrete types. The concept of the application interface was introduced, a syntax and semantics were defined for it, and its vitally important role in using box structure concepts to develop software was demonstrated. BDM enables rigorous software specification, design and verification in a way that preserves and amplifies the essence of the Box Structure Method. Furthermore, a file containing the complete ensemble of black boxes, state boxes and clear boxes can be processed by a C++ compiler.

(3) Formal Methods. BDM introduces formal methods of specification and verification not presented before. The canonical form-based function description provides a specification method for applications in which a traditional direct stimulus history-based description is not practical. The structured specification method of BDM using multi-level domain partitioning and a specification hierarchy enables the specification of a complex system in a top-down and stepwise manner which supports reuse of existing specifications. Black box verification of completeness and consistency have been introduced and facilitated for a standard form of the black box. State box verification has been significantly simplified and integrated with state migration. Clear box verification has been improved in a minor way. The method of β -expansion assures that development of all boxes stays within the envelope of BDM support.

(4) Fundamental Principles. The fundamental principles of transaction closure (completeness and consistency), referential transparency, state migration, and common services have been embodied in BDM. That is, if one performs all design and verification steps as prescribed in BDM, then one has observed these principles. BDM shifts the focus from a passive expression of principles to active achievement of the desired outcomes associated with these principles and reinforces them throughout the design process of β -expansion.

6.2 Future Work

BDM has certain weaknesses, which future work should address. In order to gain precision, it was necessary to introduce a great deal of notation and complexity. As BDM evolves it must become simpler. A thin, strong thread of advice now flows from top level black box all the way into code in C++. Additional useful patterns of expression must be developed for frequently occurring designs in all three boxes, with stronger linkages from one box form to another. Although verification has been improved, further progress is needed in making verification of boxes more expeditious and effective.

BDM is an open system which has a uniform box definition syntax and semantics. It can be augmented by a box repository of specification black boxes for various application domains. Such a repository would make software specification, design, and verification in BSM easier for systems analysts and programmers.

Hevner [50] has done some work to rationalize BSM with the object oriented paradigm. The object paradigm has a number of concepts and principles which, like BSM, are not sufficiently well defined to give adequate guidance to practitioners. Bangs [52] has explored some aspects of using BSM to prepare specifications for objects in C++. Formalization of the object paradigm within BDM would be productive.

In BDM, abstract types and variables that are close to application domains are used in black boxes and state boxes while concrete types and variables that are directly representable in C++ are used in all three box forms. Such a treatment of abstract types and concrete types makes BDM adaptable to other programming environments. Thus, another line of development would be to link other important programming languages, such as Ada (see Hopkins [41]), with BDM to make it a multi-language formal development method.

Finally, since concurrent computing and multi-tasking become ever more important, effective treatment of concurrency in box structures shall be a focus of future research.

Bibliography

- [1] Mills, H. D., R. C. Linger and A. R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, 1986.
- [2] Mills, H. D., R. C. Linger and A. R. Hevner, "Box Structured Information Systems," *IBM Systems Journal*, Vol.26, No.4, 1987.
- [3] Mills, H. D., "Stepwise Refinement and Verification in Box Structured Systems," *IEEE Computer*, June 1988, pp.23-36.
- [4] Woodcock, J. and M. Loomes, *Software Engineering Mathematics*, Addison-Wesley, 1988.
- [5] Wing, J. M., "A Specifier's Introduction to Formal Methods," *IEEE Computer*, September 1990, pp.8-21.
- [6] Meyer, B., "On Formalism in Specification," *IEEE Software*, January 1985, pp.6-26.
- [7] Scharbach, P. N., *Formal Methods: Theory and Practice*, CRC Press, 1989.
- [8] Hahl, O. J., E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972, pp.1-82.
- [9] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, 1976.
- [10] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," *Communication of ACM*, Vol. 12, No. 10, October 1969, pp. 576-583.
- [11] Hoare, C. A. R., *Notes on Data Structuring*, Academic Press, 1972, pp.83-174.
- [12] Mills, H. D., "Mathematical Foundations for Structured Programming," Technical Report FSC 71-6012, IBM Corp., 1972, also in *Software Productivity*, Little, Brown and Company, 1986, pp.115-178.
- [13] Mills, H. D., "The New Math of Computer Programming," *Communications of the ACM*, vol.18, no.1, January 1975, pp.43-48.
- [14] Parnas, D. L., "A Technique for Software Module Specification with Examples," *Communications of the ACM*, vol.15, no.5, May 1972, pp.330-336.
- [15] Bartussek, W. and D. L. Parnas, "Using Traces to Write Abstract Specifications for Software Modules," *Technical Report No.TR77-012*, University of North Carolina, December 1977.
- [16] Luckham, D. C., F. W. Henke, B. Krieg-Bruckner and O. Owe, *ANNA - A Language for Annotating Ada Programs*, Springer-Verlag, 1987.

- [17] Luckham, D., S. Sankar and S. Takahashi, "Two-Dimensional Pinpointing: Debugging with Formal Specification," *IEEE Software*, January 1991, pp.74-84.
- [18] McLean, J., "A Formal Method for the Abstract Specification of Software," *Journal of the ACM*, Vol.31, No.3, July 1984, pp.600-627.
- [19] Parnas, D. L. and Y. Wang, "The Trace Assertion Method of Module Interface Specification," *Technical Report CRL No. 244*, May 1992, McMaster University, Hamilton, Ontario, Canada L8S 4K1.
- [20] Bjørner, D. and C. B. Jonse, *Formal Specification and Software Development*, Prentice-Hall, 1982.
- [21] Jones, C. J., *System Software Development Using VDM*, Prentice-Hall, 1986.
- [22] Lafontaine, C., Y. Ledru and P. Y. Schobbens, "Two Approaches Towards the Formalisation of VDM," *VDM '90 - Formal Methods in Software Development*, Springer-Verlag, 1990.
- [23] Cohen, B., W. T. Harwood and M. I. Jackson, *The Specification of Complex Systems*, Addison-Wesley, 1986.
- [24] Spivey, J. M., *The Z Notation - A Reference Manual*, Prentice-Hall, 1989.
- [25] Hayes, I. J., *Specification Case Studies*, Prentice-Hall, 1987.
- [26] Hoare, C. A. R., "Preface," *VDM '90 - Formal Methods in Software Development*, Springer-Verlag, 1990.
- [27] King, S., "Z and Refinement Calculus," in *VDM '90 - Formal Methods in Software Development*, Springer-Verlag, 1990.
- [28] Mills, H. D., M. Dyer, and R. C. Linger, "Cleanroom Software Engineering," *IEEE Software*, September 1987, pp.19-24.
- [29] Poore, J. H. and H. D. Mills, "An Overview of the Cleanroom Software Development Process," *Formal Methods Workshop*, Halifax, Nova Scotia, July 1989.
- [30] Mills, H. D. and J. H. Poore, "Bring Software Under Statistical Quality Control," *Quality Progress*, November 1988.
- [31] Mills, H. D., "Certifying the Correctness of Software," *Proceedings of the 25th Hawaii International Conference on System Sciences*, IEEE, January 1992, pp.373-381.
- [32] Fuhrer, D., H. Mao and J. H. Poore, "OS/2 Cleanroom Environment: A Progress Report on a Cleanroom Tools Development Project," *Proceedings of the 25th Hawaii International Conference on Systems Science*, IEEE, January 1992, pp.449-458.

- [33] Gruman, G., "Major Changes in Federal Software Policy Urged," *IEEE Software*, November 1989, pp.78-80.
- [34] Staff Members, *Bugs in the Program*, House Committee on Science, Space, and Technology, US Government Printing Office, October 1989.
- [35] Parnas, D. L., "On a 'Buzzword': Hierarchical Structure," *Proc. IFIP Congress 74*, North Holland, 1974, pp.336-339.
- [36] Linger, R. C., H. D. Mills and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.
- [37] Fetzer, D. and J. H. Poore, "Using Box Structures with the Z Notation," *Proceedings of the 25th Hawaii International Conference on System Sciences*, IEEE, January 1992, pp.394-405.
- [38] Deck, M. *et al*, "Extended Semantics for Box Structures," *Proceedings of the 25th Hawaii International Conference on System Sciences*, IEEE, January 1992, pp.382-393.
- [39] Mao, H. and J. H. Poore, "CDL - A Box Structure Method Design Language for C," Department of Computer Science, University of Tennessee, Knoxville, July 1991.
- [40] Rosen, S. J., "Design Languages for Cleanroom Software Engineering," *Proceedings of the 25th Hawaii International Conference on System Sciences*, IEEE, January 1992, pp.406-417.
- [41] Hopkins, S. L., *A Box Structure Ada Design Language*, Master's Thesis, Department of Computer Science, University of Tennessee, 1990.
- [42] Ellis, M. and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
- [43] Loeckx, J. and K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons, 1987.
- [44] Mendelson, E., *Introduction to Mathematical Logic*, Madsworth & Brooks, 1987.
- [45] Gries, D., *The Science of Programming*, Springer-Verlag, 1981.
- [46] Stanat, D. F. and D. F. McAllister, *Discrete Mathematics in Computer Science*, Prentice-Hall, 1977.
- [47] Becker, S. A. and A. R. Hevner, "Concurrent System Design with Box Structures," *IEEE Proceedings of 13th Annual International COMPSAC*, Orlando, 1989, pp. 32-40.
- [48] Aho, A. V., J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

- [49] Aho, A. V., J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.
- [50] Hevner, A. R. and H. D. Mills, "Box-Structured Method for Systems Development with Objects," *IBM Systems Journal*, 1993.
- [51] Freedman, D. P. and G. M. Weinberg, *Walkthroughs, Inspections, and Technical Reviews*, Dorset House, 1990.
- [52] Bangs, A., "An Evaluation of Integration of the Trace Assertion Method with the Box Structure Method for Coding in C++," Master's Thesis, Department of Computer Science, University of Tennessee, 1993.

Appendixes

Appendix A

BDM Examples

1 Stack

1.1 Problem statement

The integer stack is a list of integers, ranging from -32766 to +32767, where all insertions and deletions take place at one end, called the top. Four operations are defined for the integer stack.

CREATE	- Create an empty stack;
PUSH x	- Insert the element x at the top of the stack;
POP	- Return and delete the top element of the stack;
TOP	- Return the element at the top of the stack.

1.2 The SH-based black box Stack

```
typedef enum {CREATE, PUSH, TOP, POP} COMMAND;

BB INT Stack( in  COMMAND Command,
              INT  Value
              out /* INT  Return */ )
{
  dom
    (Command, Value) ∈ {(CREATE, ε), (POP, ε),
                       (TOP, ε), (PUSH, n|n∈{-32766..+32767})}
  map
    def // the index of the last unpopped integer if exists
      i = Max({ j | hj=(PUSH,*) ∧ (CREATE)∉hj+1..s ∧
                  Count(__(POP),hj+1..s) = Count(__(PUSH,*),hj+1..s) })
    for
      [ Command∈{POP, TOP} ∧ i≠ε → Return = hi.Value
      || otherwise → Return = ε
      ]
}
```

1.3 The CH-based black box Stack

```

BB INT Stack( in  COMMAND Command,
               INT   Value
               out /* INT   Return */ )
{
  dom
    (Command, Value) ∈ {(CREATE, ε), (POP, ε),
                        (TOP, ε), (PUSH, n|n∈{-32766..+32767})}
  map
    def
      BB Eq(in och, s, out Return)
      {
        dom   och ∈ ({PUSH} × INT)* ∧ s ∈ COMMAND × INT
        map   def nch = Return; for
          [ s = (CREATE)      → nch = φ
            s = (PUSH, *)     → nch = och + ⟨s⟩
            s = (POP)         → nch = och/$
            s = (TOP)         → nch = och
          ]
        );
        ch = Cn(h);
        for
          [ Command ∈ {TOP, POP} ∧ ch ≠ φ → Return = ch.Value
            | otherwise                  → Return = ε
          ]
      }
}

```

1.4 The state box Stack

```

SB INT Stack( in  COMMAND Command,
               INT   Value
               out /* INT   Return */ )
{
  state
    INT* stack = φ;
  dom
    (Command, Value) ∈ {(CREATE, ε), (POP, ε),
                        (TOP, ε), (PUSH, n|n∈{-32766..+32767})}
  map
    [ Command = CREATE → Return = ε ∧ stack = φ
      | Command = PUSH  → Return = ε ∧ stack = stack' + ⟨Value⟩
      | Command = POP   →
        [ #stack' ≠ 0 → Return = stack', ∧ stack = stack'/$
          | otherwise → Return = ε ∧ stack = stack'
        ]
      | Command = TOP   →
        [ #stack' ≠ 0 → Return = stack', ∧ stack = stack'
          | otherwise → Return = ε ∧ stack = stack'
        ]
    ]
}

```

1.5 The clear box Stack

```
#include <stdio.h>
#include <stdlib.h>
#include "bdm.h"

#define NIL      -32767    // concrete representation of ε
typedef enum    {CREATE, PUSH, TOP, POP} COMMAND;
struct ITEM    {INT Value; struct ITEM *Next;};
typedef struct ITEM STACKITEM;

CB INT Stack( in  COMMAND Command,
              INT   Value
              out /* INT   Return */ )
{
    state
        static STACKITEM *pTop=NULL;
    proc
        STACKITEM *pTempItem;
        INT        Return;

        Return = NIL;
        switch(Command)
        {
            case CREATE:
                while (pTop!=NULL)
                {
                    pTempItem=pTop;
                    pTop = pTop->Next;
                    free(pTempItem);
                }
                break;
            case TOP:
                if (pTop!=NULL) Return = (pTop->Value);
                break;
            case POP:
                if (pTop!=NULL)
                {
                    Return = (pTop->Value);
                    pTempItem = pTop;
                    pTop = pTop->Next;
                    free(pTempItem);
                }
                break;
            case PUSH:
                if (Value>-32767 and Value<=32767)
                {
                    pTempItem = (STACKITEM*)malloc(sizeof(STACKITEM));
                    if (pTempItem!=NULL)
                    {
                        pTempItem->Next = pTop;
                        pTop = pTempItem;
                        pTop->Value = Value;
                    }
                    else printf("run out of memory at PUSH stack");
                }
                break;
        }
        return Return;
}
```

2 Birthday Book

2.1 Problem Statement

The Birthday Book [37] is a system to maintain a list of names (each of which may be a maximum of 64 letters) and birth dates. The intended functions of the system are as follows:

- Add a name and birth date to the system, if the name is not already in the Birthday Book.
- Delete a name and birth date from the system.
- Find the birth date for a specified name.
- Remind the user of the names of people having birthdays on a specified date.

2.2 Data Definition

```
// birthday.h  --  CDT definitions

#define  LEN_NAME  64

typedef  enum      {ADD, DELETE, FIND, REMIND} COMMAND;

typedef  CHAR      *STRING;

typedef  CHAR      *NAME;

typedef  CHAR      *NAMELIST;

typedef  struct    {UCHAR Month; UCHAR Day; USHORT Year;} DATE;

typedef  struct    {CHAR Name[LEN_NAME]; DATE  Birthday;} RECORD;

// ADT definitions

ADT STRING      CHAR*;

ADT NAME        LET*;

ADT NAMELIST    NAME*;
```

2.3 The SH-based black box BirthdayBook

```

BB BirthdayBook( in
    COMMAND    Command,
    NAME        InName,
    DATE        &InDate,
    out
    STRING      Message,
    NAMELIST    &NameList,
    DATE        &OutDate )
{
    def
        Known = { hj | hj=(ADD, Name, *) ∧ ¬NameAdded(Name, h1..j-1) ∧
                    ¬NameDeleted(Name, hj+1..s) };

        NameAdded: NAME×S+ → {TRUE, FALSE} ::=
            NameAdded(Name, L)
            = ∃i∈{1..#L}[ Li=(ADD, Name, *) ∧ ¬NameDeleted(Name, Li+1..s) ];

        NameDeleted: NAME×S+ → {TRUE, FALSE} ::=
            NameDeleted(Name, L) = (DELETE, Name, *)∈L;

    for
        [ Command=ADD      ¬
          [ (ADD, InName, *)∈Known  ¬ Message = "Name already known"
            | otherwise              ¬ Message = "Add confirmation"
          ]

          | Command=DELETE ¬
          [ (ADD, InName, *)∈Known  ¬ Message = "Delete confirmation"
            | otherwise              ¬ Message = "Name not known"
          ]

          | Command=FOUND  ¬
          [ (ADD, InName, *)∈Known  ¬ OutDate = (s.InDate | s∈Known ∧
                                                    s.InName=InName)
            | otherwise              ¬ Message = "Name not known"
          ]

          | Command=REMIND      ¬ NameList = {s.InName | s∈Known ∧
                                                    s.InDate=InDate}
          ]
    }
}

```

2.4 The CH-based black box BirthdayBook

```

BB BirthdayBook( in
                  COMMAND    Command,
                  NAME       InName,
                  DATE       &InDate,
                  out
                  STRING     Message,
                  NAMELIST   &NameList,
                  DATE       &OutDate )
{
  def
    BB Eq(in och, s, out Return)
    {
      dom   och ∈ ({ADD} × NAME × DATE)* ∧ s ∈ COMMAND × NAME × DATE

      map   def   nch = Return; for
              [ s = (ADD, Name, *) ∧ (ADD, Name, *) ∉ och
                → nch = och + {s}
              | s = (DELETE, Name, *) ∧ (ADD, Name, *) ∈ och
                → nch = och - { (ADD, Name, *) }
              | otherwise
                → nch = och
              ]

    };

    Known = Cn(h);

  for
    [ Command = ADD
      [ (ADD, InName, *) ∈ Known → Message = "Name already known"
      | otherwise                → Message = "Add confirmation"
      ]

    | Command = DELETE
      [ (ADD, InName, *) ∈ Known → Message = "Delete confirmation"
      | otherwise                → Message = "Name not known"
      ]

    | Command = FIND
      [ (ADD, InName, *) ∈ Known → OutDate = (s.InDate | s ∈ Known ∧
                                                s.InName = InName)
      | otherwise                → Message = "Name not known"
      ]

    | Command = REMIND
      → NameList = {s.InName | s ∈ Known ∧
                     s.InDate = InDate}
    ]
}

```

2.5 The state box BirthdayBook

```

SB BirthdayBook( in
    COMMAND    Command,
    NAME       InName,
    DATE       &InDate,
    out
    STRING     Message,
    NAMELIST   &NameList,
    DATE       &OutDate )
{
    state
    {STRING Name, DATE Birthday}' BirthdayFile =  $\phi$ ;
    map
    [ Command=ADD       $\rightarrow$ 
        [ (InName, *) $\in$ BirthdayFile  $\rightarrow$  Message = "Name already known"
        | otherwise       $\rightarrow$  Message = "Add confirmation" ^
                               BirthdayFile =
                               BirthdayFile' +  $\langle$  (InName, InDate)  $\rangle$ 
        ]
    | Command=DELETE   $\rightarrow$ 
        [ (InName, *) $\in$ BirthdayFile  $\rightarrow$  Message="Delete confirmation" ^
                               BirthdayFile =
                               BirthdayFile' -  $\langle$  (InName, *)  $\rangle$ 
        | otherwise       $\rightarrow$  Message = "Name not known"
        ]
    | Command=FOUND     $\rightarrow$ 
        [ (InName *) $\in$ BirthdayFile  $\rightarrow$  OutDate = (q.Birthday |
                               q $\in$ BirthdayFile ^ q.Name=InName)
        | otherwise       $\rightarrow$  Message = "Name not known"
        ]
    | Command=REMINDE   $\rightarrow$  NameList =  $\langle$  q | q $\in$ BirthdayFile
                               ^ q.Birthday=InDate  $\rangle$ 
    ]
}

```

2.6 The clear box BirthdayBook

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <io.h>
#include "bdm.h"
#include "birthday.h"

CB LONG FindRecord(in FILE *fp, NAME InName out /* Return */);
CB VOID AddRecord( in FILE *fp, NAME InName, DATE &InDate
    out /* BirthdayFile */ );

```

```

CB VOID DeleteRecord(in FILE *fp, LONG Offset  out /*BirthdayFile*());
CB VOID FindBirthday(in FILE *fp, LONG Offset, out DATE &OutDate );
CB VOID RemindList(in FILE *fp, DATE &InDate, out NAMELIST &NameList);

```

```

CB BirthdayBook( in
                  COMMAND  Command,
                  NAME      InName,
                  DATE      &InDate,
                  out
                  STRING    Message,
                  NAMELIST  &NameList,
                  DATE      &OutDate )
{
    state
        static FILE *fpBirthdayFile=NULL;
    proc
        LONG Offset;

        if (fpBirthdayFile==NULL)
            then
                fpBirthdayFile=fopen("BirthdayFile", "wb+");
                Message[0]='\0'; NameList=""; OutDate.Month=0;
                switch(Command)
                {
                    case ADD:
                        if (FindRecord(in fpBirthdayFile, InName)>=0)
                            then
                                strcpy(Message, "Name already known");
                            else
                                {
                                    AddRecord(in fpBirthdayFile, InName, InDate);
                                    strcpy(Message, "Add confirmation");
                                }
                                break;

                    case DELETE:
                        Offset = FindRecord(in fpBirthdayFile, InName);
                        if (Offset>=0)
                            then
                                {
                                    DeleteRecord(in fpBirthdayFile, Offset);
                                    strcpy(Message, "Delete confirmation");
                                }
                            else
                                strcpy(Message, "Name not known");
                                break;

                    case FIND:
                        Offset = FindRecord(in fpBirthdayFile, InName);
                        if (Offset>=0)
                            then
                                FindBirthday(in fpBirthdayFile, Offset, out OutDate);
                            else
                                strcpy(Message, "Name not known");
                                break;

                    case REMIND:
                        RemindList(in fpBirthdayFile, InDate, out NameList);
                        break;
                }
    }
}

```

2.7 Lower level boxes

```
// Return the file offset of the record that contains InName

BB LONG FindRecord( in FILE *fp, NAME InName
                   out /* LONG Return */ )
{
  def @fp=< r | r∈RECORD > for
  [ (InName, *)∈@fp → Return = (First(_=(InName,*),@fp)-1)*sizeof(RECORD)
  | otherwise → Return = -1
  ]
}

CB LONG FindRecord( in FILE *fp, NAME InName
                   out /* LONG Return */ )
{
  state
    NONE
  proc
    RECORD TempRec;

    rewind(fp);
    while(not feof(fp))
    {
      fread(&TempRec, sizeof(RECORD), 1, fp);
      if (strcmpi(InName, TempRec.Name)==0)
      then
        return ftell(fp)-sizeof(RECORD);
      }
    return (LONG) (-1);
}

BB VOID AddRecord( in FILE *fp, NAME InName, DATE &InDate
                  out /* BirthdayFile */ )
{
  def BirthdayFile = @fp = < r | r∈RECORD > for
  BirthdayFile = BirthdayFile*+<(InName,InDate)>
}

CB VOID AddRecord( in FILE *fp, NAME InName, DATE &InDate
                  out /* BirthdayFile */ )
{
  state
    NONE
  proc
    RECORD TempRec;

    memmove(TempRec.Name, InName, LEN_NAME);
    memmove(&TempRec.Birthday, &InDate, sizeof(DATE));
    fseek(fp, 0L, SEEK_END);
    fwrite(&TempRec, sizeof(RECORD), 1, fp);
}
```

```

BB VOID DeleteRecord( in FILE *fp, LONG Offset
                      out /* BirthdayFile */ )
{
    def BirthdayFile = @fp = { r | r ∈ RECORD } for
        i = (Offset/sizeof(RECORD))+1 ^
        BirthdayFile = BirthdayFile/i
}

CB VOID DeleteRecord( in FILE *fp, LONG Offset
                      out /* BirthdayFile */ )
{
    state
        NONE
    proc
        LONG FileSize, CurPosition;
        RECORD TempRec;

        FileSize = filelength(fileno(fp));
        CurPosition = Offset+sizeof(RECORD);
        // delete the record by moving each of the rest of the records
        // one record-position up
        while(CurPosition < FileSize)
        {
            fseek(fp, CurPosition, SEEK_SET);
            fread(&TempRec, sizeof(RECORD), 1, fp);
            fseek(fp, CurPosition-sizeof(RECORD), SEEK_SET);
            fwrite(&TempRec, sizeof(RECORD), 1, fp);
            CurPosition = CurPosition + sizeof(RECORD);
        }
        // decrease the file size
        chsize(fileno(fp), FileSize-sizeof(RECORD));
}

BB VOID FindBirthday( in FILE *fp, LONG Offset,
                      out DATE &OutDate )
{
    def BirthdayFile = @fp = { r | r ∈ RECORD } for
        i = (Offset/sizeof(RECORD))+1 ^
        OutDate = BirthdayFilei.Birthday
}

CB VOID FindBirthday( in FILE *fp, LONG Offset,
                      out DATE &OutDate )
{
    state
        NONE
    proc
        RECORD TempRec;

        fseek(fp, Offset, SEEK_SET);
        fread(&TempRec, sizeof(RECORD), 1, fp);
        memmove(&OutDate, &TempRec.Birthday, sizeof(DATE));
}

```

```

BB  VOID RemindList( in  FILE *fp, DATE &InDate,
                    out  NAMELIST &NameList )
{
    def BirthdayFile = @fp = { r | r ∈ RECORD } for
        NameList = { r | r ∈ BirthdayFile ∧ r.Birthday=InDate }
}

CB  VOID RemindList( in  FILE *fp, DATE &InDate,
                    out  NAMELIST &NameList )
{
    state
        NONE
    proc
        SHORT    Count;
        RECORD    TempRec;
        typedef struct ITEM {LONG offset; struct ITEM *next} LISTITEM;
        LISTITEM *pHead, *pTempItem;
        CHAR      *pNextName;

        Count = 0;
        pHead = NULL;
        rewind(fp);
        fread(&TempRec, sizeof(RECORD), 1, fp);
        // find a list of offsets of the records that match InDate
        while(not feof(fp))
        {
            if ( TempRec.Birthday.Month==InDate.Month and
                TempRec.Birthday.Day ==InDate.Day )
            then
            {
                pTempItem = (LISTITEM*)malloc(sizeof(LISTITEM));
                pTempItem->offset = ftell(fp)-sizeof(RECORD);
                pTempItem->next = pHead;
                pHead = pTempItem;
                Count++;
            }
            fread(&TempRec, sizeof(RECORD), 1, fp);
        }
        NameList = (CHAR*)malloc(Count*LEN_NAME+1);
        pNextName = NameList;
        // write names to the namelist
        while (pHead!=NULL)
        {
            fseek(fp, pHead->offset, SEEK_SET);
            fread(&TempRec, sizeof(RECORD), 1, fp);
            memmove(pNextName, &TempRec.Name, LEN_NAME);
            pNextName = pNextName+LEN_NAME;
            pTempItem=pHead;
            pHead = pHead->next;
            free(pTempItem);
        }
        *pNextName = '\0';
}

```

3 Min_Max_Avg12

Black box Min_Max_Avg12 accepts a sequence of natural numbers and produces the minimum number and the maximum number in the stimulus history, and the moving average of the last twelve stimuli in the stimulus history.

3.1 The black box Min_Max_Avg12

```
ADT PROCSTATUS {EXIT, CONTINUE, SUSPEND};
ADT NUMCHAR    {'0', '1', ..., '9'};

BB VOID Min_Max_Avg12( in  /* KEYBOARD   STRING s      */
                      out /* DISPLAY   STRING result */
                      PROCSTATUS status /* */ )
{
  dom s ∈ {"q"} ∪ NUMCHAR'

  map
    def NAT rmin, rmax; REAL ravg; for
      [ #sh=1      → result = "\nNext input: "
        | s = "q"  → result = "\nBye!\n"
        | s ∈ NUMCHAR' →
            MinMaxAvg12(in AtoN(s), out rmin, rmax, ravg) ∧
            result = PackString("\n    Output: Min=%s, Max=%s, Avg=%s",
                                NtoA(rmin), NtoA(rmax), RtoA(ravg)) + ("\n\nNext input: ")
        | otherwise →
            result = "\n    Invalid natural number\n\nNext input: "
      ] ^
      [ s = "q" → status=EXIT | otherwise → status=CONTINUE]
}

// convert an ASCII string of digits to a natural number
BB NAT AtoN(in STRING str, out Return)
{
  dom
    str ∈ NUMCHAR' ∧ str = "Dn" + "Dn-1" + .. + "D1"
  map
    Return = (Dn-'0')*10n-1 + (Dn-1-'0')*10n-2 + .. + (D1-'0')
```

```

// convert a natural number y to an ASCII string of digits
BB STRING NtoA(in NAT y, out Return)
{
  map
    n=Min({j|y<10j}) ^ i∈{1..n} [Di=(y%10i)/10i-1+'0'] ^
    Return="Dn" + "Dn-1" + .. + "D1"
}

// convert a real number y to an ASCII string of digits
BB STRING NtoA(in REAL r, out Return)
{
  dom
    r = y.x | x,y∈NAT

  map
    m=Min({j|y<10j}) ^ i∈{1..m} [Di=(y%10i)/10i-1+'0'] ^
    n=Min({j|x<10j}) ^ i∈{1..n} [Ei=(x%10i)/10i-1+'0'] ^
    Return="Dm"+"Dm-1" + .. + "D1"+"."+"En"+"En-1" + .. + "E1"
}

BB STRING PackString(in STRING str, s1, s2, ..., sn, out Return)
{
  map
    n=Count(_='%', str) ^ ∀j∈{1..n} [ij=First(_='%', str, j) ^
    Return=(str)i1, i1+1, i2, i2+1, ..., in, in+1 s1, %, s2, %, ..., sn, %
}

```

3.2 The state box Min_Max_Avg12

The black box has specified the user interface interaction and used the lower level black box MinMaxAvg12 to define computation of rmin, rmax and ravg. Since state data will be used only for producing rmin, rmax, and ravg, it can be migrated to the lower level black box leaving the top-level black box without state encapsulation. Therefore, the top-level state box is the same as the black box.

3.3 The clear box Min_Max_Avg12

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bdm.h"

typedef USHORT NAT;
typedef FLOAT REAL;
typedef enum {EXIT, CONTINUE, SUSPEND} PROCSTATUS;

#define LEN_BUFFER 81

CB VOID Min_Max_Avg12(VOID);
CB VOID Min_Max_Avg12(in NAT x, out NAT *rmin, NAT *rmax, REAL *ravg);
CB VOID main(VOID);

CB VOID Min_Max_Avg12( in /* KEYBOARD STRING s */
                      out /* DISPLAY STRING result */
                      PROCSTATUS status /* */)
{
    state
    NONE
    proc
    NAT rmin, rmax;
    REAL ravg;
    CHAR s[LEN_BUFFER];
    PROCSTATUS status=CONTINUE;
    CHAR *pRtn;

    printf("\nNext input: ");
    while(status!=EXIT)
    {
        pRtn = fgets(s, LEN_BUFFER, stdin);
        if (pRtn==NULL) /* EOF or error occurred */
            then
            {
                printf("\nEOF or an error occurred!");
                status = EXIT;
            }
        else if (strcmpi(s, "q\n")==0) /* s="q"? */
            then
            {
                printf("\nBye!\n");
                status = EXIT;
            }
        else if (strspn(s,"0123456789")==strcspn(s,"\n"))
            then /* s is a natural number */
            {
                Min_Max_Avg12(in atol(s), out &rmin, &rmax, &ravg);
                printf("\n Output: Min=%d, Max=%d, Avg=%f",
                    rmin, rmax, ravg);
                printf("\n\nNext input: ");
            }
        else
            printf("\n Invalid natural number\n\nNext input: ");
    }
}
```

3.4 The SH-based black box MinMaxAvg12

```
BB VOID MinMaxAvg12(in NAT x, out NAT *rmin, NAT *rmax, REAL *ravg)
{
    rmin=Min(sh) ^ rmax=Max(sh) ^
    ravg = [#sh≥12 → Avg(shn-11..n) | #sh<12 → Avg(sh)]
}
```

3.5 The CH-based black box MinMaxAvg12

```
BB VOID MinMaxAvg12(in NAT x, out NAT *rmin, NAT *rmax, REAL *ravg)
{
    def
    BB Eq(in och, x, out Return)
    {
        dom
        och = (<min, max>+last12 | min∈NAT^max∈NAT^last12∈NATe12 )
        map
        def nch = Return; for
        [ och=∅ → nch = <x, x, ε, ε, ..., ε, x>
        | otherwise → nch = <Min({x, och.min}), Max({x, och.max})>
        + och.last122..12+<x>
        ]
    }
    for
    ch = Cn(sh) ^
    rmin=ch.min ^ rmax=ch.max ^ ravg=Avg(ch.last12First(ε)..s)
}
```

3.6 The state box MinMaxAvg12

```
SB VOID MinMaxAvg12(in NAT x, out NAT *rmin, NAT *rmax, REAL *ravg)
{
    state
    static NAT hismin, hismax;
    static LONG last12[12]=
        {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1};
    map
    rmin=hismin=Min(x, hismin') ^
    rmax=hismax=Max(x, hismax') ^
    laset12 = last12*2..12+<x> ^
    ravg=Avg(ch.last12First(ε-1)..s)
}
```

3.7 The clear box MinMaxAvg12

```
CB VOID MinMaxAvg12(in NAT x, out NAT *rmin, NAT *rmax, REAL *ravg)
{
    state
        static NAT hismin=(NAT)-1, hismax=0;
        static LONG last12[12]=
            {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1};
    proc
        NAT sum, i, n;

        *rmin=hismin=Min(x, hismin);

        *rmax=hismax=Max(x, hismax);

        memmove(&last12[0], &last12[1], 11*sizeof(LONG));
        last12[11] = (LONG)x;

        for (i=0; last12[i]==-1 and i<12; i++);
        n=12-i;
        for (sum=0; i<12; i++) sum = sum+last12[i];
        *ravg = ((REAL)sum)/n;
    }

}

CB VOID main(VOID)
{
    Min_Max_Avg12();
}
```

4 SumOfSquareSinceReset

Black box SumOfSquareSinceReset accepts a sequence of natural numbers. A zero is considered to be a reset stimulus. The response to a stimulus history is the sum of the squares of stimuli since the last reset stimulus in the stimulus history.

4.1 The SH-based black box SumOfSquareSinceReset

```
BB VOID SumOfSquareSinceReset(in NAT x, out NAT &y)
{
    j=Last(_=0, sh)  ^
    [ j=ε → y=Σi=1, #sh (shi2)
      | j≠ε → y=Σi=j, #sh (shi2)
    ]
}
```

4.2 The CH-based black box SumOfSquareSinceReset

```
BB VOID SumOfSquareSinceReset(in NAT x, out NAT &y)
{
    def
        BB Eq(in och, x, out Return)
        { dom och∈(NAT-{0})* ^ x∈NAT
          map def nch=Return for
              [ x=0    → nch=φ
                | x≠0  → nch=och+⟨x⟩
              ]
        };
        ch=Cn(sh);
    for
        [ ch=φ → y=0
          | ch≠φ → y=Σi=1, #ch (chi2)
        ]
}
```

4.3 The state box SumOfSquareSinceReset

```
SB VOID SumOfSquareSinceReset(in NAT x, out NAT &y)
{
    state
        NAT sum=0;

    map
        [ x=0 → sum=0 ∧ y=0
        | x≠0 → sum=sum'+x2 ∧ y=sum
        ]
}
```

4.4 The clear box SumOfSquareSinceReset

```
#include "bdm.h"
typedef USHORT NAT;

CB VOID SumOfSquareSinceReset(in NAT x, out NAT &y)
{
    state
        static NAT sum=0;
    proc
        if (x==0)
            then
                y=sum=0;
            else
                y=sum=sum+x*x;
    }
}
```

Appendix B

BDM Reserved Words

ADT	and	BB	BOOL	CASE	CB
ch	CH	CHAR	Cn	def	DEFAULT
DO	dom	ELSE	Eq	FALSE	FI
for	h	IF	in	inout	INT
LET	map	NAT	NONE	not	NUM
OD	or	out	PART	proc	ran
REAL	REPEAT	Return	SB	sh	SH
state	then	THEN	TRUE	VOID	WHILE

Vita

Hailong Mao was born in Shanghai, China on August 26, 1952. He entered the Yan An Junior High School in Shanghai in 1964. His study there was interrupted by the "Great Cultural Revolution" in Fall 1966. He then worked for seven years as a farmer, a band member, and an electrician at a state-owned farm in Jiangsu Province, about 200 miles north of Shanghai. He attended East China Normal University in Shanghai between 1975 and 1982. He received a diploma for undergraduate study in Physics in 1978 and was awarded the Master of Engineering with a major in Computer Science in 1982.

After working as an instructor for two and half years in the Department of Computer Science at East China University of Chemical Technology, Shanghai, he was invited by renowned computer scientist and educator Dr. Raymond Yeh to Austin, Texas and worked as a software engineer at his company Syscorp International from June 1985 to December 1987.

He entered the graduate program at the University of Tennessee, Knoxville, in January 1988. He was awarded the Master of Science in 1990 and the Doctor of Philosophy in 1993, both in Computer Science. He received Chancellor's Citation for Extraordinary Professional Promise in April 1992.

The author is currently employed as an Sr. Advisory Programmer by IBM Corporation, Charlotte, North Carolina.