To the Graduate Council:

I am submitting herewith a thesis written by Daniel T. Fetzer entitled "Using Box Structures with the Z Notation." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.

J. H. Poore, Major Professor

We have read this thesis
and recommend its acceptance:

Jean R. S. Blair

Michael D. Moore

Ronald P. Lemmi

Accepted for the Council:

Associate Vice Chancellor
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Daniel T. Fetzer

Date December 1991

USING BOX STRUCTURES WITH THE Z NOTATION

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Daniel T. Fetzer
May 1992

DEDICATION

In memory of
Rachel,
Winnie Mae,
Charles,
Harold,
Lucille and
Evelyn.

ACKNOWLEDGEMENTS

Several scholars and professionals have assisted me generously in the course of this research. I appreciate Dr. Jesse Poore for the opportunity to participate in his software quality research seminars, for suggesting this topic, and for his excellent guidance of this research. I thank my committee—Professors Jean Blair, Ronald Leinius, and Michael Vose—for their careful review and and their helpful comments on this thesis. I also wish to thank Dr. Bill McClain who provided me with excellent classroom instruction. The software quality seminar provided an excellent forum to discuss and develop these ideas. I appreciate the helpful comments and encouragement of Mitchel Duitz, Dave Fuhrer, Scott Hopkins, Hailong Mao, Carment Trammel, and Jim Whittaker. I am especially grateful for the invaluable TEX and LATEX assistance provided by Robert McGaffey, Karen Barry and Judi Theg. My final thanks go to my wife Beth, who gave me a continuous supply of love, patience, and encouragement even when I seemed to be spending more time working on this project than I did with her.

ABSTRACT

The Box Structure Method (BSM) of analysis and design provides a framework that can be used to introduce formality into the requirements specification stage of software development. A method of requirements specification is presented which integrates the Z notation with BSM. The requirements specification is confined to the top level black box specification and the corresponding top level state box specification. We describe criteria for good requirements specification and we explain advantages of our integrated method for achieving them. Summary introductions are given to both BSM and Z followed by an examination of the relationship between the two methods. We explain our integrated method and illustrate it using a simple birthday reminder system.

This method has been used to give a mathematical model of a tool called the Box Structure Assistant (BSA). The Box Structure Assistant is a planned system to provide tools for managing, editing, and printing system designs using the C Design Language. The BSA black box and state box requirements specifications are presented in a case study. Finally, we discuss issues that have emerged from the use of this method and indicate areas for future research.

TABLE OF CONTENTS

CHAPTER 1

Introduction

This paper reports on our experience in integrating the Box Structure Method (BSM) [18] with the Z notation [24] for specifying system requirements. We are limiting the scope of this paper to specifying system requirements of a problem to be solved as opposed to designing a solution.

We propose developing a two stage specification: a top level black box specification followed by a state box specification. The black box and the state box describe identical external behavior. The black box behavior is described using stimulus histories, while the state box behavior is described by referencing an internal system state.

The black box view represents the system as a device that accepts a stimulus and produces a response before accepting the next stimulus. There are no references to internal state data or implementation procedures. The response is uniquely determined by the system's *stimulus history*—the sequence of stimuli previously accepted by the system.

The state box utilizes the Z notation to represent state data and state box transition rules. The Z notation is a formal specification language developed at the University of Oxford and is based on set theory and predicate calculus. State data are encapsulated from the black box and represented using abstract data types such as sets, relations, functions and sequences. The Z notation is used in the state box for representing state data, invariant conditions, and state box transition rules. Transition rules for required operations are expressed implicitly in terms of pre- and post-conditions.

This integrated method is used to present a requirements specification for the Box Structure Assistant. The Box Structure Assistant (BSA) is a tool intended to assist a designer in creating, editing, viewing and printing box structured designs.

For the purpose of specifying requirements, we have found that using the Z notation with BSM has advantages over using either method alone. By defining the black-box behavior first, we postpone a commitment to specific state data until we understand the state data requirements. The black box view allows us to focus on *what* needs to be done, not *how* to do it. Using Z notation to describe the state box allows us to introduce state data at an abstract level. The precisely defined syntax and semantics of the Z notation provide a level of formality needed to verify correct designs and implementations.

This paper is organized as follows:

- In chapter 2, we will describe what we consider to be good characteristics of a requirements specification and indicate methods to achieve them.
- In chapter 3, a brief overview of the Box Structure Method is given.
- In chapter 4, an introduction to the Z notation is provided.

- In chapter 5, we explore how BSM and Z are related and explain our method for integrating them.
- Chapter 6 provides an example.
- Chapter 7 gives an introduction to the Box Structure Assistant case study.
- Chapter 8 is the black box specification of BSA.
- Chapter 9 is the state box specification of BSA.
- In Chapter 10, we summarize the proposed method and consider directions for future work.

CHAPTER 2

What is a good specification?

The requirements specification is an important artifact of the software development life cycle. It serves as a basis for communication among various interested parties of a software system. Between the client and the specifier, the specification serves as a contract and facilitates an understanding of what is to be done. For the designer, the specification forms a basis for the development and verification of system designs. For the programmer, the specification may be used to validate the implementation. For the tester, the specification is used to prepare test cases and to evaluate their executions. For the documenter, the specification is a starting point for preparing user guides and reference manuals. What characteristics should a requirements specification have to effectively support these various roles?

2.1 What, not how

Most will agree that a specification should describe *what* is to be done, not *how* to do it. Unfortunately, there is less agreement on the division between *what* and *how*. In general, *what* refers to a specification that concisely and precisely describes the functional requirements and nothing more. That is, the specification should avoid bias toward any possible implementation. This implies that the requirements specification should avoid procedural descriptions, explicit data structures, and unnecessary early commitment to other implementation details. The requirements specification should be confined to defining the problem to be solved, not to describing possible solutions.

Explicit data structures may be avoided by using history-based specification methods. These methods specify behavior by using transition rules on stimulus histories [18,17], or by using logic assertions [2,9] on sequences of access program calls.

Procedural descriptions may be avoided by using functional mappings or implicit specification methods. Implicit specifications use predicate calculus to specify pre-conditions that input arguments must satisfy and post-conditions that outputs must also satisfy [13,24].

Ideally, the requirements specification should be independent of a particular human-computer interface or of a particular operating system and hardware platform. The concept of *dialogue independence* suggests that we separate the specification of the dialogue from the computational components of the software.

> *Dialogue independence* is an approach in which design decisions affecting only the human-computer dialogue are isolated from those affecting only application system structure and computational software. In practice this means

that the appearance of the interface to the end-user and choices of interaction styles (e.g., command languages, menus, forms) used to extract inputs from the end-user are not known to the computational software. [7]

2.2 Correctness

The most important characteristic of a good specification is that it be correct. Errors committed in the specification stage of software development are often the most difficult to detect and costly to remove. Specification errors are not likely to be discovered until very late in the life cycle or after the system has been delivered to the customer. Yet, these early stages of development are rarely handled in a disciplined manner. Formal methods can help eliminate errors at this early stage by making the specification more precise. Hall gives this account of using formal methods to eliminate errors:

> In an informal specification, it is hard to tell what is an error, because it is not clear what is being said. When challenged, people try to defend their informal specification by reinterpreting it to meet the criticism. With a formal specification, we have found that errors are much more easily found—and, once they are found, everyone is more ready to agree that they are errors. [6]

2.3 Clear communication

A specification is a medium of communication between the specifier and the customer. Since the customer must validate the specification, it is important that the specification be understandable by the customer. Natural language specifications can impair this communication since they can be verbose, vague, contradictory, incomplete and ambiguous. On the other hand, mathematical specifications can help improve this communication in a number of ways. Mathematical notations are precise and concise with unambiguous and well-defined definitions. It is easier to reason in mathematics. Formal specifications allow a specifier to systematically develop the theory of the application domain and to ask deeper questions about the properties of the system than would be possible with a natural language specification.

Formal specifications are often criticized as requiring too many mathematical skills for sponsors and users to understand. However desirable for sponsors to read and validate formal specifications, it is impractical to expect this level of involvement from most customers. Several actions may be taken to mitigate these concerns.

- Provide a separate external specification to the user that paraphrases the formal internal specification in natural language. Cobb and Mills proposed using a three part specification which includes an expected-usage profile along with an external and internal specification [3].
- Use formal reasoning to demonstrate the consequences of the specification. When a

specification is expressed formally, one can use formal reasoning to demonstrate to the user that the specification meets certain requirements.

- Animate the specification. Build prototypes to explore requirements issues and record the results in the formal specification. Some specification methods are executable giving a direct prototyping capability.
- Provide training in discrete mathematics and the formal notation. For example, the Z notation requires a background in set theory and predicate calculus.

CHAPTER 3

The Box Structure Method

BSM organizes a system design into a usage hierarchy of abstractions. Each abstraction consists of a black box, state box and clear box representation. Each of these three box structures defines identical external behavior, but with increasing levels of internal visibility.

3.1 The black box

The black box represents each abstraction as a function mapping the current stimulus and the stimulus history to a response.

$$bb : S \times \text{seq}\, S \rightarrow R$$

We use S to represent the set of all possible stimuli, $\text{seq}\, S$ to represent the set of all possible sequences of S, and R to represent the set of possible responses. We can use the Box Design Language (BDL) to define this function. BDL was introduced as a "language for recording, communicating, and analyzing box structures among developers, users, and managers of information systems." [18]

The BDL organizes a black box into three parts: stimulus, response, and transition.

define BB ⟨*BB name*⟩
  stimulus
    ⟨*stimulus*⟩ : ⟨*type*⟩
    ⋮
  response
    ⟨*response*⟩ : ⟨*type*⟩
    ⋮
  transition
    ⟨*transition rule*⟩
    ⋮

The stimulus section identifies the domain set of stimuli and the response section identifies the range set of responses. Each stimulus or response is described by listing its name and data type. The ⟨*type*⟩ specification identifies a set of permissible values from which the stimulus may be drawn. The transition section is used to define transition rules between stimuli and responses. Each transition rule defines a mapping between a stimulus-response pair by defining the stimulus histories that are valid for that mapping.

An advantage of using stimulus histories to define the stimulus-response mappings is that it avoids a premature commitment to invented state data abstractions.

In BDL, the stimulus history of a black box has the form

$$\langle stimulus \rangle.0,\ \langle stimulus \rangle.1,\ \ldots$$

where $\langle stimulus \rangle.0$ names the current stimulus and $\langle stimulus \rangle.i$ refers to the ith stimulus prior to the current stimulus. For example, the black box for a simple device that displays the sum of the current stimulus and the previous stimulus can be defined as follows:

define BB Add2
  stimulus
    $S : \mathbb{N}$
  response
    $R : \mathbb{N}$
  transition
    $R := S.0 + S.1$

This box accepts stimulus S of type $\mathbb{N}$ ($\mathbb{N}$ denotes the set of natural numbers) and, for each stimulus received, produces a response named R which is equal to the sum of the current stimulus ($S.0$) and the previous stimulus ($S.1$).

BSM allows transition rules to be expressed at any level of formality, as Mills stated for the black box function f:

> For especially simple data abstractions, f can be given in closed mathematical notation. For large abstractions, it may be necessary to give f in the natural language of a problem domain, often a mixture of formal and informal language. Whatever the notation, the black box description is a function. [17]

3.2 The state box

The state box view of the system differs from the black box view by introducing state data abstractions to encapsulate and replace the stimulus histories. A special and simple form of the state box is a finite-state machine derived from the black box. The stimulus history of the black box can be regarded as the state. But the black box is not a finite-state machine since the stimulus histories are unbounded. The state box can also be represented as a function which maps stimuli and old states to responses and new states. Let T represent the set of all possible states. The state machine is given by the function sb as follows:

$$sb : S \times T \rightarrow R \times T \ ^{\dagger}$$

This special form of a state box will be sufficient for the specification problem.

The BDL syntax organizes the state box into four parts: stimulus, response, state, and machine.

[†] For simplicity, we adopt the view of the classical state machine instead of the state box function in [17] ($g : S^* \times T^* \rightarrow R \times T$) whose domain contains stimulus and state histories.

define SB ⟨*SB name*⟩
stimulus
⟨*stimulus*⟩ : ⟨*type*⟩
⋮
response
⟨*response*⟩ : ⟨*type*⟩
⋮
state
⟨*state data*⟩ : ⟨*type*⟩
⋮
machine
⟨*SB transition rule*⟩
⋮

The stimulus and response sections of the state box match the stimulus and response sections of the black box. This equivalence formally implies the following predicate:

$$\begin{aligned} \mathrm{dom}\,(\mathrm{dom}\,bb) &= \mathrm{dom}\,(\mathrm{dom}\,sb) \wedge \\ \mathrm{rng}\,bb &= \mathrm{dom}\,(\mathrm{rng}\,sb) \end{aligned}$$

That is, the set of stimuli that can be recognized by the black box is equal to the set of stimuli that can be processed by its corresponding state box. Further, the set of possible responses that are produced by a black box is equal to the set of possible responses that can be produced by its corresponding state box.

The state box differs in two ways from the black box. First, state data are identified and recorded to meet requirements implied by the stimulus history based transition rules of the black box. Secondly, the transition rules of the state box must be revised to refer to the state data instead of the stimulus histories.

In illustration, the Add2 black box can be expanded into the following state box:

define SB Add2
stimulus
$S : \mathbb{N}$
response
$R : \mathbb{N}$
state
$L : \mathbb{N}$
machine
$R := S + L$
$L := S$

In this example, the state variable L is introduced to record the previous stimulus. Both the black box and state box representations accept the same set of stimuli and produce the same set of responses. But the state box transition of Add2 refers to the state variable L instead of the stimulus history $S.1$.

While the black box is a *procedure-free* and *state-free* description of external behavior, the state box expansion is a *procedure-free* but *state-defined* description of the same behavior. Thus, state data of the state box must encapsulate the unbounded set of stimulus history into a finite set of internal states that lets a finite-state machine mimic the black box behavior. The state box expansion introduces multiple design possibilities since there are multiple state boxes that can satisfy the behavior required by a single black box. However, there is only one black box that behaves like a given state box.

3.3 The clear box

BSM also provides a clear box view in which control structures are introduced to describe the processing of the stimulus and state [17]. The clear box can introduce new black boxes allowing a decomposition of the system into a usage hierarchy of modules. The general form of the state box permits state data to be pushed down into the new black boxes. For purposes of requirements specification, we do not use the clear box view since it introduces algorithmic descriptions and imposes a particular module decomposition that goes beyond specifying *what* needs to be done. BSM can be used to design a solution where use of the clear box is appropriate, but that is outside the scope of this paper.

One disadvantage of avoiding the clear box view, is that we lose one method of subdividing the problem into manageable pieces via stepwise refinement. To help compensate for this loss, we define the stimuli and responses at an abstract level. For purposes of requirements specification, we restrict our attention to the functions required of the software as we attempt to avoid bias toward any possible implementation. For example, instead of defining our stimulus set as every keystroke typed by the user, we define the stimuli as a set of functional commands. This method of abstraction will be described further in chapter 5.

CHAPTER 4

The Z notation

The Z notation was developed during the early 1980's by the Programming Research Group at the Oxford University Computing Laboratory. The original work includes papers by Abrial [1], Sufrin [26], and Morgan [20]. We follow the notation as defined in the reference manual by Spivey [24].

The Z notation is a mathematical notation for expressing and communicating the specifications of computer programs. Z uses conventional notations of logic and set theory organized into expressions called *schemas*. Mathematical data types (such as sets, relations, sequences and functions) are used to model the data in a system. The use of these abstract data types suppresses implementation details and allows us to use mathematical laws to reason about them. The notation of *predicate calculus* is used to implicitly describe the effect of each operation on the system. A brief summary of the set theory and predicate calculus notations is provided in Appendix A.

4.1 Z schemas

A *schema* is a named unit consisting of a *signature* (a declaration of variables) together with a predicate relating the variables of the signature. A schema S may be represented either horizontally

$$S \mathrel{\widehat=} [s_1 : T_1; s_2 : T_2; \cdots; s_n : T_n \mid f_1 \wedge f_2 \wedge \cdots \wedge f_m]$$

or vertically

$$\begin{array}{|l} S \\ \hline s_1 : T_1 \\ s_2 : T_2 \\ \cdots \\ s_n : T_n \\ \hline f_1 \\ f_2 \\ \cdots \\ f_m \\ \hline \end{array}$$

where $s_i : T_i (i = 1 \ldots n)$ is the *signature* and the conjunction of $f_i (i = 1 \ldots m)$ represents the *predicate* of the schema. The set of $s_i (i = 1 \ldots n)$ are called variables and the $T_i (i = 1 \ldots n)$

are types.

Each f_i represents a first order formula on the variables in the declaration part. (Note that in the vertical form of the schema the logical *and-connective* ($\wedge$) separating each f_i is omitted, but implied when no other connective is present.)

4.2 States and operations

Schemas may be used to describe both static and dynamic aspects of a system. The static aspects are described by *state space* and *initial state* schemas while dynamic aspects are described by *operation* schemas. In a *state space schema* the signature is used to describe the state data of the system. Invariant conditions that must be maintained by the operations of the system are given by the predicate.

An *initial state schema* has the same signature as a corresponding state space schema. The predicate part is used to give initial values to the state data.

Each *operation schema* can introduce four kinds of variables in their declaration part:

1. input variables (decorated (suffixed) by '?'),
2. state data variables before the operation (no decoration),
3. output variables (decorated by '!'), and
4. state data variables after the operation (decorated by '').

The relationship among the *input, before-state, output* and *after-state* variables is given by the predicate. Preconditions specify acceptable values for the *input* and *before-state* variables. Postconditions specify how the *output* and *after-state* variables will be affected by the operation provided the precondition holds.

4.3 Schema inclusion and decoration

A previously defined schema S may be included into a new schema T by listing S in the declaration part. In this case the declarations of S and T are merged and their predicates are conjoined.

If a schema name is suffixed with a decoration, then all the components of the schema are suffixed with the same decoration. For example, if S is a schema, then S' is the same as S, except all of the component names are suffixed with the decoration $'$.

Schema inclusion and decoration are illustrated in sections 5.1 and 6.2.

4.4 The Δ and Ξ conventions

The Δ and Ξ conventions make the introduction of *before-state* and *after-state* variables into an operation schema more convenient. If *State* is a previously defined state

space schema, then $\Delta State$ is implicitly defined as the combination of $State$ and $State'$, unless a different definition is explicitly given:

$$\Delta State \triangleq State \land State'$$

The introduction of $\Delta State$ into an operation schema implies that the state data may be changed by the operation.

For operations that only access state data without changing them, $\Xi State$ is implicitly defined as

$$\Xi State \triangleq [\Delta State \mid \Theta State = \Theta State']$$

The expression $\Theta State = \Theta State'$ in the predicate records the equality relationship between the corresponding before and after state variables.

4.5 Axiomatic descriptions

An axiomatic description appears in the form

$$[Declaration \mid Predicate; \cdots; Predicate]$$

where the *declaration* part is used to introduce one or more global variables. Any constraints on these variables are listed in the *predicate*. Axiomatic definitions are in effect for the remainder of the specification.

An example of an axiomatic description is given by the following definition of the *min* function.

$$[min : \mathbb{P}_1 \mathcal{Z} \pfun \mathcal{Z} \mid min = \{S : \mathbb{P}_1 \mathcal{Z}; m : \mathcal{Z} \mid m \in S \land (\forall n : S \bullet m \leq n) \bullet S \mapsto m\}]$$

4.6 Schema operations

Schemas can be combined in various ways to produce new schemas. The most frequently used schema operators are *conjunction* ($\land$) and *disjunction* ($\lor$).

Two *type compatible* schemas S and T may be *conjoined* by the expression $S \wedge T$. Two schemas are said to be type compatible if every variable common to both schemas has the same type in each signature. This operation has the effect of merging their declarations and conjoining their predicates.

Two *type compatible* schemas S and T may be *disjoined* by the expression $S \vee T$. This operation has the effect of merging their declarations and disjoining their predicates.

Examples of these schema operations will be provided in subsequent chapters.

CHAPTER 5

Z and the Box Structure Method

In this section we will first explore the relationship between Z and box structures by using the Z notation to model a black box. We will then explain our approach to integrating the two methods. The reader is advised to refer to the previous chapter and to the appendix for unfamiliar notations.

5.1 Modeling box structures with Z

The Z notation can be used to give an abstract model of the Box Structure notions of stimulus, response, stimulus history, and black box transitions for a given system. First we define sets containing all responses, commands, and parameters as *basic types* of our model.

$$[RESPONSE, COMMAND, PARAMETER]$$

The use of *basic types* allows us to name a set of objects without having to specify the details of those objects. We define *STIMULUS* to be an abbreviation for the *Cartesian product* of *COMMAND* and a sequence of parameters:

$$STIMULUS == COMMAND \times \text{seq}\, PARAMETER$$

By modeling a stimulus as a parameterized command (instead of as a single token or keystroke), we hope to achieve a level of abstraction that is appropriate for the specification of requirements.

We can now represent the concept of a *Stimulus_History* with a schema:

Stimulus_History

$$stim\_hist : \text{seq}\,(STIMULUS)$$
$$mrs : STIMULUS$$

$$stim\_hist \neq \langle\rangle \wedge mrs = last\ stim\_hist \vee$$
$$stim\_hist = \langle\rangle \wedge mrs = undefined$$

The schema consists of the schema name (*Stimulus_History*) followed by declarations of variables before the dividing line, and a predicate part below the line which gives a relationship among the values of the variables. The stimulus history of our system (*stim_hist*) is declared as a sequence of stimuli. The most recent stimulus (*mrs*) is defined as the *last* stimulus in *stim_hist* if *stim_hist* is not equal to the empty sequence ($\langle\rangle$). Otherwise, *mrs* is *undefined.*

The concept of a black box for a system may also be represented by a schema:

Black_Box

$$known\_stimuli : \mathbb{P}\, STIMULUS$$
$$known\_responses : \mathbb{P}\, RESPONSE$$
$$traces : \mathbb{P}\,(\text{seq}\, STIMULUS)$$
$$transition : \text{seq}\, STIMULUS \nrightarrow RESPONSE$$

$$known\_stimuli = \text{rng}\,(\text{dom}\, transition)$$
$$known\_responses = \text{rng}\, transition$$
$$traces = \text{dom}\, transition$$

$\mathbb{P}$ indicates that the variables *known_stimuli* and *known_responses* each contain a set of values. The *known_stimuli* contains the set of all stimuli recognized by our system. The set *known_responses* represents all possible responses that may be produced. The set *traces* represents all the possible sequences of stimuli that are recognized by our system. The concept of a *transition* is represented by a function which maps a sequence of *stimuli* to a *response*. The invariant condition records the fact that the sets *known_stimuli*, *known_responses*, and *traces* can be derived from the transition function.

We use the following initial state schema to specify that the *stim_hist* is equal to the empty sequence when the system is first started.

Init_Stimulus_History

$$Stimulus\_History$$

$$stim\_hist = \langle\rangle$$

The first declaration is a reference to the schema *Stimulus_History*. When a schema name is referenced as a declaration, it includes all the declarations and predicates of the referenced schema into the current schema. Thus the variable *stim_hist* is defined because of the reference to the schema *Stimulus_History*.

The concept of a black box transition can be represented by the following operation schema:

BB_transition

$$\Xi Black\_Box$$
$$\Delta Stimulus\_History$$
$$current\_stimulus? : STIMULUS$$
$$response! : RESPONSE$$

$$stim\_hist \frown \langle current\_stimulus? \rangle \in traces$$
$$stim\_hist' = stim\_hist \frown \langle current\_stimulus? \rangle$$

$response! = transition(stim\_hist')$

The reference to the $\Xi Black\_Box$ schema indicates that its components are accessed, but not changed by the black box transition operation. The $\Delta Stimulus\_History$ reference introduces $stim\_hist, stim\_hist', mrs$ and mrs' into the schema. The operation accepts as input the *current_stimulus*? and produces the output *response*!. A pre-condition to the operation is that the *current_stimulus*? concatenated to the *stim_hist* must be an element in *traces*. If so, this concatenation becomes the new *stim_hist'* and the *response*! is defined by applying the function *transition* to the argument *stim_hist'*.

For this model, we are assuming that the data in the *Black_Box* schema has been pre-defined. Of course, defining the sets of *known_stimuli*, *known_responses*, and the *transition* function is the task of the specifier in developing a black box specification. The *known_stimuli* and *known_responses* sets can be defined either by explicitly naming their members or by giving rules (such as a formal grammar) for defining them. It is impossible, in general, to enumerate all the members of the *transition* function since the stimulus histories may be unbounded. Instead, this function may be characterized by defining a partition $\Pi : \mathbb{P}(I \nrightarrow \mathbb{P}\,\mathrm{seq}\,STIMULUS)$ of the domain of *transition* such that all elements within any given member of Π will map to the same response in *transition*. In Z, we can state this idea as follows:

$$\Pi \textbf{ partitions } \mathrm{dom}\, transition \mid$$
$$\forall t : \Pi \bullet \forall i, j : \mathrm{rng}\, t \bullet$$
$$transition(i) = transition(j)$$

Our organization of the black box transition rules, as described in the next section, is based upon this partition.

5.2 Specification method

The subsections that follow explain our approach for using Z and Box Structures together to develop a requirements specification. The specification is developed in two distinct and complementary stages. In the first stage the black box functionality is specified using a combination of natural language and formal notations. The specification is organized by stimulus-response pairs in a way that facilitates a rigorous analysis of the specification. The correctness of the specification must be validated by the customer. The black box specification forms a basis for the development of the second stage state box specification which is written using the formal Z Notation.

5.2.1 Specifying the black box

Our goal in specifying the black box *transition* function is not to achieve full formality, but to organize the specification in a way that will conveniently allow various levels of formality to coexist. The requirements specification is a communication medium which evolves during the requirements analysis process. It is subject to many changes as new facts are made available and as the application domain is better understood. We want to be able to formalize parts of a system without having to model the entire system. A change to formalize one part of a specification should have minimal impact on the rest of the specification.

Our black box specification is divided into the standard three sections of *stimulus*, *response*, and *transition*, but we make changes within each of these sections to handle requirements specification.

define BB ⟨*BB name*⟩
  stimulus
    ⟨*stim_defn*⟩
    ⋮
  response
    ⟨*response_defn*⟩
    ⋮
  transition
    ⟨*transition rule*⟩
    ⋮

We define a stimulus to be a command followed by an optional list of parameters. Each parameter may be assigned a type.

⟨*stim_defn*⟩ ::= ⟨*command*⟩ [⟨*param_defn*⟩ ⋯]
⟨*param_defn*⟩ ::= ⟨*var_name*⟩ [: ⟨*type*⟩]

The *response* section lists what is known about desired responses of the system. This section describes output displays, reports and user messages. BNF (for Backus-Naur Form) may be used to specify the syntax of these outputs. These descriptions should focus on the content of the information to be displayed or printed, not its exact appearance.

The black box *transition* section describes functional responses to current stimuli in terms of stimulus histories. This section is organized by listing all known stimuli and their transition rules.

transition
  ⟨*stimulus1*⟩
    $\mathcal{V}1$: ⟨*valid_histories1.1*⟩
    $\mathcal{R}1$: ⟨*response1.1*⟩
    $\mathcal{V}2$: ⟨*valid_histories1.2*⟩

$\mathcal{R}2$: ⟨*response1.2*⟩
⋮
⟨*stimulus2*⟩
$\mathcal{V}1$: ⟨*valid_histories2.1*⟩
$\mathcal{R}1$: ⟨*response2.1*⟩
⋮

For each current stimulus $\mathcal{S}_i (i = 1..n)$, we list all known responses in subsections labeled $\mathcal{R}_j (j = 1..m)$. Each $\mathcal{R}_j$ section is preceded by a subsection labeled $\mathcal{V}j$: where $\mathcal{V}j$: lists conditions to identify the set of legal histories (traces) that lead to the response identified in $\mathcal{R}j$:. This organization defines the transition function in the form of a *conditional rule*:*

$$
\begin{aligned}
(&\mathcal{S}_1 \wedge \mathcal{V}_{1.1} \rightarrow \mathcal{R}_{1.1} \mid \mathcal{S}_1 \wedge \mathcal{V}_{1.2} \rightarrow \mathcal{R}_{1.2} \mid \cdots \\
&\mathcal{S}_2 \wedge \mathcal{V}_{2.1} \rightarrow \mathcal{R}_{2.1} \mid \mathcal{S}_2 \wedge \mathcal{V}_{2.2} \rightarrow \mathcal{R}_{2.2} \mid \cdots \\
&\qquad \vdots \\
&\mathcal{S}_n \wedge \mathcal{V}_{n.1} \rightarrow \mathcal{R}_{n.1} \mid \cdots \mid \mathcal{S}_n \wedge \mathcal{V}_{n.j} \rightarrow \mathcal{R}_{n.j})
\end{aligned}
$$

This rule can be read as "if $\mathcal{S}_1$ and $\mathcal{V}_{1.1}$ then $\mathcal{R}_{1.1}$; else if $\mathcal{S}_1$ and $\mathcal{V}_{1.2}$ then $\mathcal{R}_{1.2}$; else ...; if $\mathcal{S}_2 \wedge \mathcal{V}_{2.1}$ then $\mathcal{R}_{2.1}$; ... ."

Formal assertions in the black box

The black box can be expressed at any level of formality or using a mixture of different levels of formality. The Z notation can be used to make formal assertions about valid stimulus histories (*traces*). The abbreviation *mrs* stands for the *most recent stimulus* and *stim_hist* represents the *stimulus history* sequence. The expression *stim_hist* = $\langle\rangle$ states that *stim_hist* is equal to the null-sequence. This means no valid stimuli have been received since the system was first initialized. As defined in [27], the relations $\hookleftarrow$ ("prefixes"), $\hookrightarrow$ ("suffixed by"), and $\overset{\cdots}{\hookrightarrow}$ ("contains") are useful for describing constraints on traces as is the function *Prefixes* which maps a sequence to all its prefixes.

*A *conditional rule* as defined in [16] consists of a sequence of (predicate $\rightarrow$ rule) pairs separated by vertical bars ($p_1 \rightarrow r_1 \mid p_2 \rightarrow r_2 \mid \cdots \mid p_k \rightarrow r_k$). For the first predicate, p_i, which evaluates to *true*, if any, the rule r_i is used.

$$
\begin{array}{|l}
\hookleftarrow : \operatorname{seq} E \leftrightarrow \operatorname{seq} E \\
\hookrightarrow : \operatorname{seq} E \leftrightarrow \operatorname{seq} E \\
\overset{\cdots}{\hookrightarrow} : \operatorname{seq} E \leftrightarrow \operatorname{seq} E \\
\mathit{Prefixes} : \operatorname{seq} E \rightarrow \mathbb{P} \operatorname{seq} E \\
\hline
s \hookleftarrow t \Leftrightarrow (\exists r : \operatorname{seq} E \bullet s \frown r = t) \\
t \hookrightarrow s \Leftrightarrow (\exists r : \operatorname{seq} E \bullet r \frown s = t) \\
t \overset{\cdots}{\hookrightarrow} s \Leftrightarrow (\exists r, r' : \operatorname{seq} E \bullet r \frown s \frown r' = t) \\
\mathit{Prefixes}\ t = s : \operatorname{seq} E \mid s \hookleftarrow t
\end{array}
$$

For example, if we let *Trace* represent a set of traces, then to make the assertion "Every *delete k* must be preceded at some stage by an *insert k*"

$$\forall t : \mathit{Trace} \bullet t \hookrightarrow \langle \mathit{delete}\ k \rangle \Rightarrow \exists p : \mathit{Prefixes}\ t \bullet p \hookrightarrow \langle \mathit{insert}\ k \rangle$$

It would be possible to express a black box specification entirely in a closed mathematical notation such as Z. In practice, however, we find it more convenient to specify the black box using a combination of natural language and formal notations. If the specifier intends to use the black box specification as a communication tool with the user, the level of formality must be limited to what the user can understand.

5.2.2 Specifying the state box

The top level state box requires the addition of the *state*, and *machine* sections to our specification.

state
  ⟨*state_schema1*⟩
  ⟨*state_schema2*⟩
  ⋮
machine
  ⟨*operation_schema1*⟩
  ⟨*operation_schema2*⟩
  ⋮

Once the black-box behavior has been defined, state data requirements are identified by examining those responses that are dependent on stimulus histories. State data are defined and recorded using Z state space schemas in the *state data* section.

The *machine* section is to record the state box transition rules using initial state and operation schemas. These new schemas can be derived from the black box transition rules but reference the state space schemas defined in the *state* section instead of the stimulus

history. An initial state schema specifies the inital state of the system along with invariant conditions that must be maintained by each operation. An operation schema is developed for each black box transition rule as follows. The parameters of the stimulus command become input variables to the schema. The response variables produced by the black box from the stimulus command become the output variables of the schema. Each black box conditional statement (labeled $\mathcal{V}_i$) corresponds to the pre-conditions of the operational schema. Black box commands that produce *delayed effects* cause updates to the state data. A delayed effect means that a command will alter the responses produced by the subsequent execution of other black box commands.

CHAPTER 6

The birthday reminder example

Spivey [24] uses an example of a birthday book system to illustrate Z notation concepts. We have used a similar birthday card system in classroom exercises to illustrate the Box Structure Method. This simple problem is presented below to illustrate how the Z notation can be integrated with box structures.

The birthday reminder is a system to maintain a list of names and birth dates, and to issue reminders when the day arrives. The functions required by the system are as follows:

- Add a name and birth date to the system.
- Delete a name and birth date from the system.
- Find the birth date for a specified name.
- Issue reminders of names of people having birthdays on a specified date.

6.1 Design the black box

According to the problem statement, we will need to deal with names and dates. We also assume various information messages will be displayed to the user. At this stage we can suppress the detail of the exact form these names, dates, and messages take by declaring them as *basic types* of our specification.

$$[NAME, DATE, MESSAGE]$$

To shorten the length of our transition rules, certain states of our stimulus history are abbreviated. We will use the phrase "name known to system" to stand for "the name was previously added to the system and was not subsequently deleted". The phrase "name not known to system" will stand for "the name was not previously added unless it was subsequently deleted".

define BB Birthday Reminder

stimulus

InitSystem
Add (name: NAME, date: DATE)
Delete (name: NAME)
FindBirthday (name: NAME)
Remind (today: DATE)

response

$\langle add\_confirm \rangle$: MESSAGE
$\langle already\_known \rangle$: MESSAGE
$\langle delete\_confirm \rangle$: MESSAGE
date : DATE
$\langle not\_known \rangle$: MESSAGE
reminders : $\mathbb{P}\, NAME$

transition

1. InitSystem
   V1: *true*
   R1: null response ($stim\_hist = \langle\rangle$)
2. Add (name: NAME, date: DATE)
   V1: Name not known to system
   R1: $\langle add\_confirm \rangle$
   V2: Name known to system
   R2: $\langle already\_known \rangle$
3. Delete (name: NAME)
   V1: Name known to system
   R1: $\langle delete\_confirm \rangle$
   V2: Name not known to system
   R2: $\langle not\_known \rangle$
4. FindBirthday (name: NAME)
   V1: Name known to system
   R1: date a birth date is displayed which is the date entered for the name
   V2: Name not known to system
   R2: $\langle not\_known \rangle$
5. Remind (today: DATE)
   V1: *true*
   R1: reminders Display every *name* known to system such that the *birthday* corresponding to the *name* is equal to *today.*

Note that *reminders,* the response to the *Remind* command, has a power set data type $\mathbb{P}\, NAME$. Thus the result of this command is a possibly empty set of names with birthdays corresponding to the specified date.

6.2 Design the state box

The next step is to determine the state data needed to encapsulate the stimulus histories. The transition rules for the *Add, Delete* and *FindBirthday* commands all indicate

that we must represent what *names* are known to the system. The response $\mathcal{R}1$: to the FindBirthday command tells us that we must be able to associate at most one birth date for a specified name. But the response to the Remind command indicates that we may need to associate several names with a given date. These observations lead us to the following representation of our state data.

state

BirthdayBook

$$known : \mathbb{P}\ NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\ birthday$$

In this *state space* schema we represent the state data with the variables *known* and *birthday*. The variable *known* is a set of names with birth dates recorded and *birthday* is a function which maps names to birth dates. By describing the variable *birthday* as a function we have implicitly asserted that each person can have at most one birthday, and that two people can share the same birthday. The invariant condition states that the variable *known* can be derived from the value of *birthday*. It would be possible to specify the system without any reference to *known* at all. However, giving a name to the concept *known* makes the specification more readable. Since we are describing an abstract view of state, we are not committed to explicitly representing *known* in an implementation.

Notice we have avoided implementation details, such as placing a limit on the number of birthdays that can be recorded, or specifying the exact format of the names and dates.

From the *MESSAGE* responses that can be elicited from our system, we will use the following *free type* definition to define *MESSAGE* as a set containing exactly these four values:

$$MESSAGE ::= add\_confirm \mid delete\_confirm \mid already\_known \mid not\_known$$

We can now turn our attention to the state box transition rules which we will represent as Z schemas in the *machine* section of the state box.

machine

1. InitSystem

InitSystem

$$BirthdayBook$$

$$known = \emptyset$$

This schema specifies the *initial state* of the system. It describes state of the system before any stimuli have been received. The predicate says that the set *known* is empty and—as a consequence of the *BirthdayBook* invariant—the *birthday* function is empty too.

2. Add

We will construct a specification for the *Add* command using two subschemas: *AddSuccess* and *AlreadyKnown*.

AddSuccess

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$
$$result! : MESSAGE$$

$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$
$$result! = add\_confirm$$

The declaration $\Delta BirthdayBook$ lets us know that the *AddSuccess* operation will change the state data. Note that the input parameters *name* and *date* correspond to the parameters for the black box *Add* command. The state box transition rule is given by the predicate below the central dividing line. The first line gives a *precondition* that the name must not be known to the system. Note how this directly corresponds to the *V1:* black box condition for the *Add* command. If the precondition is true, the second line says that the birthday function will be extended to map the new name to the given date. The third line says that the user will receive a confirmation message which corresponds to the *R1:* black box response.

The *AddSuccess* schema only handles the *V1:* black box condition. For the *V2:* black box condition we introduce the *AlreadyKnown* schema.

AlreadyKnown

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$result! : MESSAGE$$

$$name? \in known$$
$$result! = already\_known$$

The $\Xi BirthdayBook$ declaration specifies that the state of the system will not change. Note

that the precondition corresponds to the $\mathcal{V}2$: condition of the *Add* black box command, and the postcondition corresponds to the $\mathcal{R}2$: response of that command.

Using the Z schema calculus we can combine these two schemas into one robust version of the *Add* command:

$$Add \mathrel{\widehat=} AddSuccess \lor AlreadyKnown$$

Add is formed from schemas *AddSuccess* and *AlreadyKnown* by merging their declarations and disjoining their predicates. This new definition of *Add* will handle properly both the $\mathcal{V}1$: and $\mathcal{V}2$: conditions listed in the black box transition rule.

3. Delete

The operation to delete a name is specified using subschemas *DeleteSuccess* and *NotKnown*.

DeleteSuccess

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$result! : MESSAGE$$

$$name? \in known$$
$$birthday' = \{name?\} \lhd birthday$$
$$result! = delete\_confirm$$

To delete the entry for *name*? it must be in the system as specified by the first line of the predicate. The resulting function *birthday'* is set equal to the original *birthday* function with *name*? deleted from its domain.

To specify the $\mathcal{V}2$: and $\mathcal{R}2$: transition rule for the *Delete* command we introduce the *NotKnown* schema.

NotKnown

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$result! : MESSAGE$$

$$name? \notin known$$
$$result! = not\_known$$

The robust delete schema can now be specified as

$$Delete \mathrel{\widehat=} DeleteSuccess \lor NotKnown$$

4. FindBirthday

The *FindBirthday* black box command is specified using the subschema *FindBirthday-Success* and the previously defined *NotKnown* subschema.

FindBirthdaySuccess

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$date! : DATE$$

$$name? \in known$$
$$date! = birthday(name?)$$

The inclusion of $\Xi BirthdayBook$ indicates that the state data does not change as a result of this operation. If the precondition is met, the output *date*! is the value of the birthday function at argument *name*?.

The robust version of *FindBirthday* uses the *NotKnown* schema as follows:

$$FindBirthday \mathrel{\widehat=} FindBirthdaySuccess \lor NotKnown$$

5. Remind

The *Remind* black box command can be specified with one schema.

Remind

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$reminders! : \mathbb{P}\, NAME$$

$$reminders! = \{n : known \mid birthday(n) = today?\}$$

This operation has one input *today*?, and one output, *reminders*!, which is a set of names. The output *reminders*? is specified to be equal to all values n drawn from the set *known* such that the value of the birthday function at n is *today*?.

CHAPTER 7

The Box Structure Assistant

In this chapter we will explain the background, objectives and informal system requirements for the Box Structure Assistant (BSA) design tool.

7.1 Background

BSA is based upon an earlier effort, the Box Structure BDL Generator, as specified in [22]. The specification for that program used a state machine format for specifying the human-computer interface (the *dialogue component*), but used informal text for describing the system functions (the *computational component*). The ambiguity and incompleteness in the natural language descriptions led to problems during system design and verification activities. The experience from that development led us to investigate the use of formal specification notations for specifying the system functionality.

7.2 Specification Summary

The following subsections summarize functional, performance and interface requirements of the BSA.

7.2.1 Functions

The purpose of the Box Structure Assistant is to provide a software design tool to support the development and management of software designs written in the C Design Language (CDL). BSA reinforces the stepwise refinement of the Box Structure design process by imposing the methodology on the users of the tool. BSA must support the following general capabilities:

- Structure editor for entering and editing CDL designs
- Display of CDL designs
- Printing CDL designs
- Maintain project designs in a library
- Support reusable design libraries
- Print Usage Hierarchy representations

7.2.2 Performance

BSA must meet the following performance criteria:

Correctness The resulting software must be validated to ensure it meets all the requirements specified in this document.

Reliability The system must consistently perform its intended function with a high degree of reliability. A reliability goal must be established and the software must be certified to meet the goal through the process of statistical testing.

Efficiency The system must efficiently use computing resources, but not at the expense of maintainability, usability or flexibility software quality factors.

Usability The effort required to learn, operate, prepare input, and interpret output should be minimized by providing appropriate user instructions, on-line help, consistent user prompts, validation of inputs, facilities for user correction of inputs, and appropriate error messages. The Box Structured Assistant must adhere to IBM's Common User Access standards.

Maintainability The effort required to locate and fix software errors will be minimized by meeting the criteria of consistency, simplicity, modularity, and self-descriptiveness. Consistency implies uniform design and implementation techniques and notation. Simplicity implies practices such as top down design, structured programming, and limiting complexity of modules. Modularity implies that the resulting software be composed of independent objects. Self-descriptiveness implies that the resulting software should contain adequate comments, meaningful variable names, and adhere to indentation and formatting standards. The criteria of consistency, simplicity, modularity, and self-descriptiveness are all supported by the Box Structure Method.

Testability The effort required to test the system to ensure that it performs its intended functions will be minimized by adhering to modular design principles, limiting module complexity and utilizing defensive programming techniques.

Flexibility The time required to enhance system capabilities will be minimized by a modular design and by attention to future requirements.

Reusability Modules will be developed in such a way that they can be reused in other applications or on different systems. Criteria related to reusability includes generality, modularity, software system independence, machine independence and self-descriptiveness.

7.2.3 Interface Specifications

The following subsections summarize the specifications for the user, hardware and software interfaces.

User Interfaces

The user interface standards as specified by IBM's *Common User Access: Advanced Interface Design Guide* [10] and the *OSF/Motif Style Guide* [21] should be followed as closely as allowed by the terminal hardware capabilities. The standards described in *Common User Access: Basic Interface Design Guide* [11] should be followed for providing support for nonprogrammable terminals.

Software Interfaces

- Indirect interface to TEX [14] by allowing designs to be output in a TEX source file format.
- The software should run under either OS/2 or DOS 3.0 and higher. However, operating system dependencies should be hidden and localized to allow for the possibility of porting to other operating systems (such as UNIX).

Hardware Interfaces

- Primary platform shall be for an IBM PC, however, the code should be developed for the possibility of porting to other hardware platforms (e.g., SUN and DEC workstations).
- The system should be compatible with either EGA or VGA monitors in color mode. However, display dependencies should be localized to support future portability to other display devices ranging from VT220 style devices to X-windows terminals.
- The software should provide *mouse* support according to the IBM CUA standards if a mouse is available. All functions of the system must continue to be available from the keyboard interface for users who choose not to use the mouse interface.
- Allow printing compatibility with both dot matrix (Epson) and laser (HP) printers. Localize specific printer dependencies to printer configuration files.

7.3 Functional Requirements

The following subsections will describe each functional requirement of BSA in terms of its purpose, and the general capabilities that are desired or required.

7.3.1 Structure design editor

Purpose

Traditionally programs and design specifications are written and manipulated with text editors. Most programs and designs however are not purely text. In particular, a CDL

design is a hierarchically structured collection of design objects including black boxes, state boxes, clear boxes, stimulus-response lists, data types, clear box procedures and various procedural constructs. An editor that allows the user to directly create and manipulate a program or design in terms of these objects or language constructs is called a syntax-directed, language-sensitive or structured editor. The reasons for providing such an editor with the BSA include the following:

- Reduce typing effort and minimize the errors that may be introduced by typing mistakes
- Guarantee that generated designs are syntactically correct
- Allow editing operations to be performed by referencing the CDL design constructs (e.g., move to a selected black box, delete the *if-then-else* construct at the cursor).
- System knowledge of the design structure will allow more of the design process to be supported and automated within a design development environment.

Of course, BSA must also provide many standard text editing operations (e.g., data descriptions or black box transition rules). Also certain standard parts of the design shall be entered using form based entry techniques such as fill-in-the-blank for variable names or selecting a data type from a pick list (icon menu).

Requirements

A Box Structure design shall be defined as follows:

⟨*Design_Unit*⟩ ::= ⟨*Black_box*⟩
⟨*State_box*⟩
⟨*Clear_box*⟩
[⟨*Design_Unit*⟩]

That is, a Box Structure design shall be considered a tree of design units where each design unit consists of a *black_box* followed by a *state_box* followed by a *clear_box*. The first design unit, called the top level design box (level 0), shall be considered the root node of the tree. The clear box of the root node may reference n other design units by *use BB* statements. Each of these BB references defines level 1. Each design unit, except for the top level box, must be previously referenced by a *use BB* statement within the clear box of its parent design unit.

⟨*Black_Box*⟩ ::= ⟨*Stimulus*⟩
⟨*Response*⟩
⟨*Transition*⟩

⟨*State_Box*⟩ ::= ⟨*State*⟩
⟨*Machine*⟩

⟨*Clear_Box*⟩ ::= ⟨*Data*⟩
⟨*Procedure*⟩

Any one design unit shall be organized into the following sequence of design parts where the user is allowed to view or manipulate up to two design parts at a time:

Design_part_sequence :: ⟨*Stimulus*,
Response,
Transition,
State,
Machine,
Data,
Procedure⟩

Design Navigation Move operations must allow the user to easily position the displayed window among the various design units (boxes) and parts within the design as follows:

Move to next part This command allows the user to move the displayed window to the next part in the sequence.

Move to next design unit This command causes the black box of the next design unit to be displayed. The visitation order will be the same order in which boxes have been *opened* during a session.

Move to previous part Move the display window to the previous design part in the current design part sequence.

Move to previous design unit Select the last part in the design part sequence of the previous design unit. The visitation order for *move to previous design unit* should be the reverse order of the *move to next design unit* operation.

In addition to having operations for moving to previous or next design parts, the user should also be provided with commands for moving directly to a design part within the current design unit and/or moving to other design units within a project design.

Move to selected design part This command will allow the user to move to a selected design part within the current design unit. The user selects the desired part for viewing from an icon menu *(pick list)*.

Move to selected design unit This command allows the user to select a desired design unit for viewing from an icon menu.

Invariant conditions and rules:

- A *move to next part* followed by *move to prev part* should result in no change in position.
- In general, an operation followed by its *mirror* operation should result a final state that matches the original state.
- *Move* commands do not change the "content" of the design.

Design manipulation Other commands that operate on design units include the following:

Change box name User is allowed to rename a black box design unit.

Delete box User is allowed to delete a black box and all lower level successors.

The design parts should be presented to the user with the following kinds of interactive forms:

data_item_form This form allows the user to enter a list of data name, description, data type triples. (e.g., ⟨*BB_Stimulus*⟩, ⟨*BB_Response*⟩, ⟨*CB_Machine_Data*⟩ parts have this type). This is a composite object used by a user to add, change or delete variable names, types and variable descriptions (comments). The commands available to the user should include:

Insert ⟨*char*⟩ Insert characters into a field on the form.

Move[⟨*left* | *right*⟩][⟨*char* | *field* | *record*⟩] Move left or right by units of characters, fields or records.

Move[⟨*up* | *down*⟩][⟨*record* | *window*⟩] Move up or down by units of records or by a window full of data.

Delete[⟨*left* | *right*⟩][⟨*char* | *field* | *record*⟩] Delete the character, field, or record to the immediate left or to the immediate right. The immediate right in practice would actually be the character, field or record where the cursor is currently positioned. Deletion to the left of fields and records are not required capabilities.

Pick_Item[⟨*abbreviation* | *cursor*⟩] Allow the user to select items from an associated pick list to fill-in fields on the form.

text_editor This form includes standard text editing forms (e.g., transition rules, data and type descriptions). The commands available to the user here should include the following:

Move[⟨*left* | *right*⟩][⟨*char* | *word* | *line* | *text_area*⟩] The user must be able to move left or right by units of characters, words, lines, or by the entire text area (for moving to the top or bottom of a text area).

Insert ⟨*char*⟩ Characters may be inserted into the text area.

Delete[⟨*left* | *right*⟩][⟨*char* | *word* | *line* | *text_area*⟩] The user must be able to delete the character, word, line, or text_area on or to the right of the user's cursor or to the left of the user's cursor.

icon_menu The user is presented with a list of items (a *pick list*) where one of the items is highlighted. Three main operations on this form: *Prev_item* positions light bar to previous item, *Next_item* positions to next item, and *select* command selects the current highlighted item for processing.

syntax_editor Used for CB Procedure. User is allowed to construct a CB procedure using pick lists for CDL procedural constructs. The *syntax_editor* also inherits many of the operations of the *text_editor* object.

A user creates references to lower level black boxes by inserting *Use BB* statements into the clear box procedure.

If the specified box already exists, the user will be presented a template for the calling arguments. For each argument in the called routine, if a variable exists in the calling routine with the same name and type, then that variable will be inserted automatically. The user may enter any remaining arguments, with the program validating that the data types are correct.

If the user inserts a *Use BB* reference to a non-existent box, then the new box is created. The user must specify the stimulus-response interface to the new box and specify the project library where the box will reside.

7.3.2 Display of CDL designs

Purpose

The user is provided the ability to view a complete design unit (BB-SB-CB) within a display-only scroll window. This capability will allow the user to verify the complete design unit in a format that will match the printed copy. The display should use indentation and key word bolding to improve readability.

Requirements

The following commands should be provided to allow the user to view the design.

Scroll line forward	IF the last line of design unit is visible THEN message "You are already at the bottom" ELSE scroll forward one line
Scroll line backward	IF the first line of the design unit is visible THEN message "You are already at the top" ELSE scroll backward one line
Scroll window forward	IF the last line of the design unit is visible THEN message "You are already at the bottom" ELSE move the last line in the window to the top and display subsequent lines

Scroll window backward	IF the first line of the design unit is visible THEN message "You are already at the top" ELSE scroll the first line in the window to the bottom of the window and display preceding lines
Move to top	IF the first line of the design unit is visible THEN message "You are already at the top" ELSE display the first n lines of the design unit in the scroll window where n is the length of the window.
Move to bottom	IF the bottom line of the design unit is visible THEN message "You are already at the bottom" ELSE display the last line of the design unit on the last line of the scroll window fill in the preceding lines of the design unit.
Search	This command will allow the user to search for text strings in the design box in either a forward or backward direction. If a search is successful, the cursor will move to the new location and a portion of the design surrounding the cursor will be displayed in the scroll window.

7.3.3 Printing CDL designs

CDL designs may be printed to either a laser printer, a dot-matrix printer or to a text file as either normal text or in a TEX source file format. Indentation and key word bolding should be used to enhance readability. The user should have the option of printing page numbers and line numbers as an aid for code verification. Options should be provided to list a summary of the lower-level boxes used by a design box and to list all modules that use a specified box.

7.3.4 Print Usage Hierarchy

Provide an option to print an indentation chart showing the usage hierarchy of the system. The relative hierarchy numbers for each box should be listed.

7.3.5 Librarian capabilities

A system should support cooperative design work by allowing an individual to check-out modules for extended work. No other users will be able to work on modules that are

currently checked out. The user must be able to later turn in the module and re-integrate the module into the design. When this is done, the calling interface of the turned in module and all uses of lower level black boxes by the module are verified.

CHAPTER 8

The BSA Black Box Specification

In this chapter, we present the black box specification for the Box Structure Assistant using the integrated method that we described earlier. The Box Structure Assistant (BSA) is intended to assist a designer in creating, editing, viewing and printing box structured designs. This document specifies the required functions of the Box Structure Assistant without attempting to specify the exact "look and feel" of the human-computer interface. BSA is specified first using a black box description. We attempt to avoid specific data representations or procedures.

This black box specification consists of a stimulus section, a response section, and a transition section. The definitions in the next section explain concepts needed for understanding the subsequent specification. The stimulus section defines stimuli the software will act upon and the response section lists responses produced by the system. The transition section defines each response in terms of a stimulus history.

8.1 Definitions

This subsection defines concepts needed for understanding the subsequent Box Structure Assistant specification.

box Refers to one entire design unit (black box, state box, clear box).

box library A *box library* is simply a collection of boxes associated with a project.

clipboard (user term) A user hold area that can store data for transfer from one part of the BSA application to another. Data in the clipboard is available to other applications.

cursor (user term) A visual cue that indicates the user's current position within a BDL design document. The portion of the BDL design that is referenced by the *cursor* appears on the users display.

dialogue box A window in which users provide information that is required by an application to perform a user request.

part Refers to one part or component of a box design unit. Each box consists of the following sequence of parts: ⟨*black box stimulus-response, black box transition rule, state box data, state box machine, clear box data, clear box procedure*⟩ .

project A *project* consists of a *box library* and a *project structure*.

project structure A *project structure* is a description of how a collection of boxes are related to one another in a usage hierarchy.

structure Refers to design language syntax structures.

8.2. define **BB** Box Structure Assistant

8.3. stimulus

We have organized related stimulus commands into groups. The commands in each group are generally related by being part of the same submenu or because they perform operations on the same kind of conceptual object. The following stimuli represent operations on projects, boxes and box parts.

1. Begin Session

Commands to allow a user to begin a BSA session.

Invoke_System User invokes BSA application.

Action_Select User selects an action to be performed from the Action_Menu. Choices include: ⟨*Box* | *Edit* | *Project* | *Part* | *Structure* | *Options* | *Help*⟩.

2. Box Operations

The commands in this group allow the user to create, retrieve, modify, delete, print and retrieve design boxes.

Box Invoke the Box_Menu. User selects one of the following actions to be performed: ⟨*New* | *Open* | *Save* | *Rename* | *Delete* | *Print* | *Next_Box* | *Prev_Box* | *Exit*⟩

New Create a new box.

Open Open a specified box.

Save Save the contents of the currently open box.

Rename Change the name of a box.

Delete Delete a box.

Print box [*and children*] to
⟨display
| ⟨*laser*|*dot matrix*⟩ printer
| ⟨*standard* | *TEX*⟩ file⟩
The box operations of ⟨*Next_Box*⟩ and ⟨*Prev_Box*⟩ should be mapped directly to a user gesture (e.g., a key press or a particular mouse action).

Next_Box Open the next box in the design.

Prev_Box Open the previous box in the design.

Exit Command to exit the current BSA session.

3. Edit Menu Commands

The edit menu allows users to reorganize CDL design documents by cutting, copying, and pasting sections of a design box to other locations within the same box or to other design boxes. Since a CDL design is a *structured document*, care must be taken to ensure that every *cut*, *copy* or *paste* action will maintain a valid design structure.

Edit Invoke the Edit_Menu. User selects one of the following actions to be performed: ⟨*Undo* | *Cut* | *Copy* | *Paste*⟩.

Undo *Undo* reverses the most recently executed user action of ⟨*Paste*⟩, ⟨*Cut*⟩, or the *Structure* menu's insert or modify construct commands.

Cut Copies the selection to the clipboard and removes it from the object being edited. The space occupied by the removed text is compressed.

Copy Copy a selection to the clipboard without removing it from the object being edited.

Paste Copy the contents of the clipboard into the object being edited.

4. Project Operations

Commands in this group allow a user to create, print or manipulate project libraries. Commands are also provided to allow a user to reserve and replace boxes into the project libaries.

Project Invoke the Project_Menu. User selects one of the following actions to be performed: ⟨*New* | *Open* | *Delete* | *Print* | *Reserve* | *Unreserve* | *Replace*⟩.

New Create a new project.

Open Open an existing project.

Delete Delete an existing project.

Print Print a project hierarchy chart for the current project to
⟨*display* |
⟨*laser* | *dot matrix*⟩ *printer* |
⟨*standard* | *TEX*⟩ *file*⟩

Reserve Reserve a module from the project library.

Unreserve Unreserve a module from the project library without replacing it.

Replace Replace a reserved module into the project library.

5. Part Operations

This group of operations allows a user to move from one part of a black box to another.

Part Invoke the Part_Menu. User selects one of the following actions to be performed: ⟨*Next_Part* | *Prev_Part* | *Stimulus* | *Response* | *Transition* | *State* | *Machine* | *Data* | *Procedure*⟩.

Next_Part Move to next part in design sequence.

Prev_Part Move to the previous part in the design sequence.

Part operations of ⟨*Next_Part*⟩ and ⟨*Prev_Part*⟩ should also be mapped to particular user gestures (keystroke or mouse action) to allow the user to bypass this menu. (For example, *accelerator* keys should be provided as a fast way to select these choices.)

The following commands allow the user to move the cursor to a specified design part within the current box.

BB Stimulus Move to the black box stimulus design part.

BB Response Move to the black box response part.

BB Transition Move to the black box transition part.

SB Data Move to the state box data part.

SB Transition Move to the state box transition part.

CB Machine data Move to the clear box machine data part.

CB Procedure Move to the clear box procedure part.

6. Structure Menu Operations

The structure menu operations are available for inserting or modifying CDL syntax structures when the user is editing the CDL clear box procedure part of a design unit.

Structure Invoke the structure menu. This pull-down menu lists the following submenus: ⟨*Insert* | *Modify*⟩. Note that the edit menu allows the user to delete structures.

Insert Invoke an item menu listing CDL syntax structures from which the user may choose one to insert into the CDL clear box procedure at the current cursor position. The defined structures that may be inserted before a CDL statement in the clear box include ⟨*ifthen*⟩, ⟨*case*⟩, ⟨*whiledo*⟩, ⟨*repeatwhile*⟩, ⟨*use_BB*⟩, and ⟨*C_statement*⟩. When the user picks a structure, a corresponding structure template is inserted directly into the current cursor position. The cursor is then moved to the first non-terminal placeholder within the inserted template.

Modify Invoke an item menu listing commands that may be used to modify the syntax structure at the current cursor position. For example, if the cursor is positioned over the ⟨*if*⟩ token of an ⟨*ifthen*⟩ statement, the possible *Modify* menu commands would include: ⟨*delete*⟩, ⟨*sequence*⟩, ⟨*case*⟩, ⟨*whiledo*⟩, and ⟨*repeatwhile*⟩. The ⟨*sequence*⟩ command would convert the *ifthen* structure to a sequence by removing the outer syntax tokens (⟨*if*⟩, ⟨*then*⟩, and ⟨*fi*⟩) and the conditional expression of the *ifthen* statement but, would leave the embedded statements within the ⟨*then*⟩ and ⟨*else*⟩ clauses. The ⟨*delete*⟩ command would remove the entire structure between the ⟨*if*⟩ and matching ⟨*fi*⟩ tokens. The ⟨*case*⟩, ⟨*whiledo*⟩, or ⟨*repeatwhile*⟩ command would convert the *ifthen* structure into a *case*, *whiledo*, or *repeatwhile* statement, respectively.

7. Options Menu

The options menu is intended to allow the user to customize the application in various ways.

New Structures Palette This option allows the user to keep a display on the screen listing all available syntax structure choices. This display only appears while the user is editing a clear box procedure.

Modify Structures Palette This option allows the user to keep a display on the screen listing all possible commands for modifying the syntax structure at the current users position.

Visitation Rules Invoke a sub-menu to specify options for box visitation order.

8. Help Menu

The user should be provided with the following options for requesting help at any point in the application.

Context Help This command should be used for requesting specific information about the item at the current cursor position.

Extended Help Command to request information about the tasks that can be performed in the current application window.

Help Help Request for information about how to use the help facility.

Keys Help Request information about key assignments within the BSA application.

9. Data Item Form Operations

The *data item form* is a generic form used to allow user entry and editing of data definitions such as the stimulus, response, state and data box parts. Each *data item form* consists of a sequence of *records*, each record consists of a sequence of *fields* and each field consists of a sequence of characters. A *record* contains a definition of one data item. The *fields* in a record identify the data item and provide a list of attributes for the item. The following general operations are available on each *data item form*.

Insert ⟨*character*⟩ Insert a character into a field at the current cursor position.

Overstrike ⟨*character*⟩ Type over a character in a field at the current cursor position.

Ins_Ovr_Toggle Command to switch the current mode—either from *Insert* mode to *Overstrike* mode or from *Overstrike* mode to *Insert* mode.

⟨*left* | *right*⟩ *Move* ⟨*character* | *field* | *record*⟩ A family of operations to allow the user to move the cursor in either direction from one character, field or record to the next one.

⟨*left* | *right*⟩ *Delete* ⟨*char* | *field* | *record*⟩ A family of operations to allow the user to delete the previous or next character, field or record.

Open ⟨*record*⟩ Command to open a record slot before the current record.

Pick_item An item menu is invoked for entering a field value by selecting from a menu.

10. Text Editor Operations

The text editor is a generic form used for editing text in transition rules, data descriptions and procedural comments.

Insert ⟨*char*⟩

Overstrike ⟨*char*⟩

Ins_Ovr_Toggle Command to switch the current mode—either from *Insert* mode to *Overstrike* mode or from *Overstrike* mode to *Insert* mode.

⟨*left* | *right*⟩ *Move* ⟨*char* | *word* | *line* | *text_area*⟩

⟨*left* | *right*⟩ *Delete* ⟨*char* | *word* | *line* | *text_area*⟩

11. Item Menu Operations

Item menus are used in the *pull-down* menus to allow the user to select from a small fixed set of choices.

Prev_item Position light bar to the previous item.

Next_item Position light bar to the next item.

Mneumonic Position light bar to item by its typing its mneumonic letter.

Abbreviate Position light bar to the next choice that starts with a specified character.

Select Select the currently highlighted item

Add_Item Add a new item to a variable length item list.

12. Box Display Operations
This set of operations allows the user to view a box design unit.

Move ⟨*line* | *page* | *part* | *box*⟩⟨*forward* | *backward*⟩

Search ⟨*text_phrase*⟩ ⟨*forward* | *backward*⟩

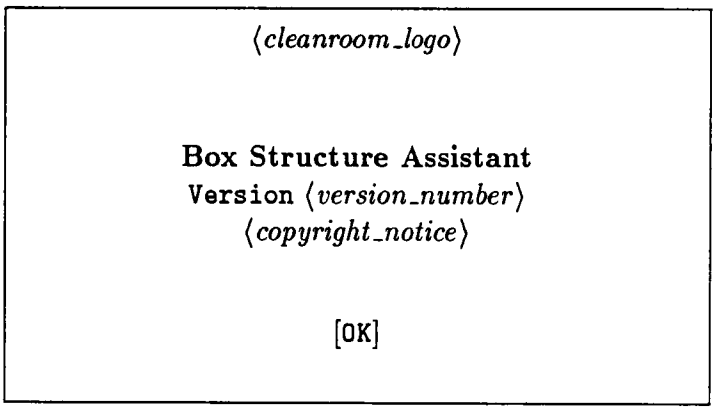
8.4. response

The response section describes all required responses to be produced by the BSA system.

8.4.1 Display Forms

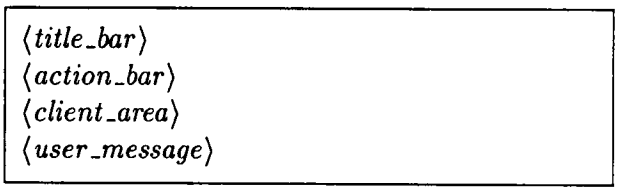
The following display forms may appear on the user screen at various points in the processing:

Welcome_Form The welcome form is displayed the first time users start the BSA application. The welcome form is removed automatically after 3 seconds or when the user selects the OK pushbutton.



Primary_Window Primary window for the Box Structure Assistant. This is the main focal point for user operations. The title bar shows the application name (Box Structure Assistant) and the name of the current project. On initial entry to the BSA the name of the project is *Untitled.*

⟨*Primary_Window*⟩ ::=



⟨*title_bar*⟩ ::=

BSA Project: ⟨*current_project*⟩ Box: ⟨*current_box*⟩

⟨*current_project*⟩ ::= *current_project : PROJ_ABBR*
Here we declare the response variable *current_project* to be of data type *PROJ_ABBR* from which its values are drawn. This variable will be used to show the user the abbreviation (acronym) for the current project which is being accessed.

⟨*current_box*⟩ ::= *current_box : BOX_NAME*
The name of the current box being edited.

⟨*action_bar*⟩ ::=

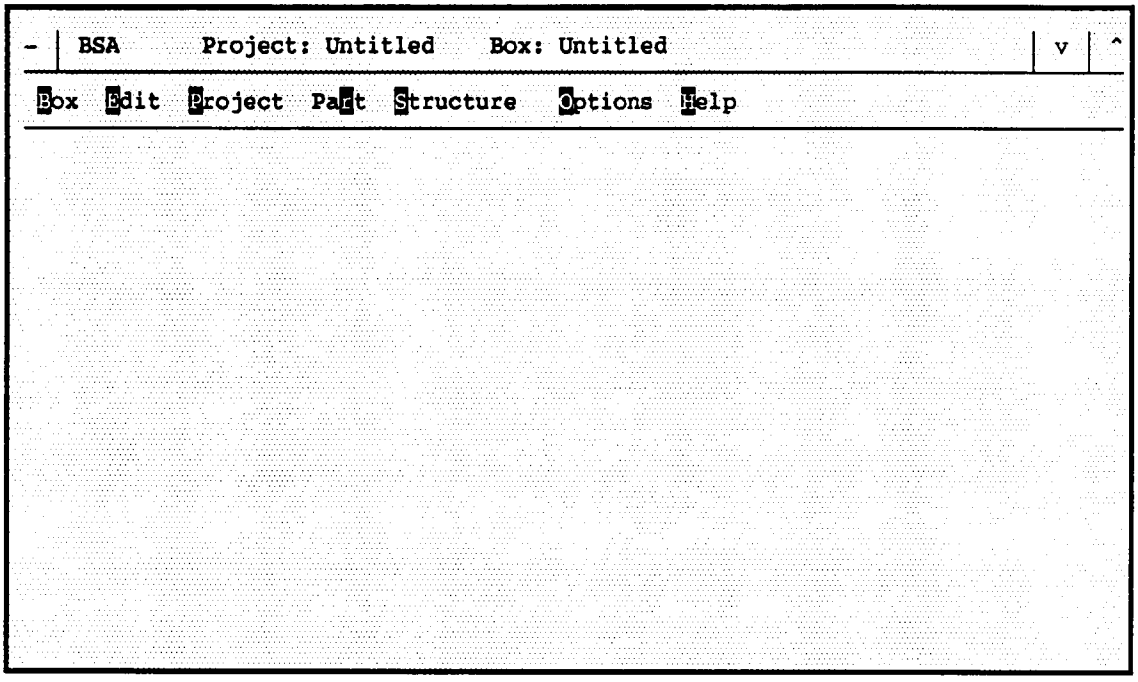
Box Edit Project Part Structure Options Help

⟨*user_message*⟩ ::= *user_message : MESSAGE*
A one line area at the bottom of the screen for displaying confirmation and warning messages to the user. Designers may also elect to implement a *message box* as defined in [10] for displaying these messages.

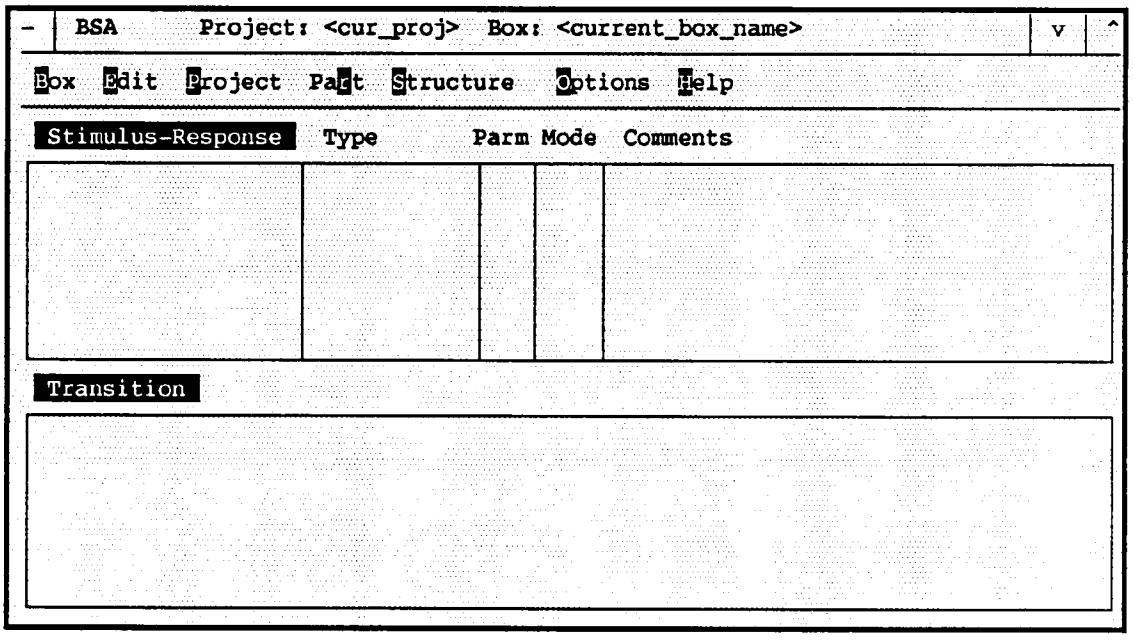
⟨*client_area*⟩ ::= ⟨*blank*⟩ |
⟨*stimulus_response*⟩ |< *transition* >|
⟨*state*⟩ | ⟨*machine*⟩ |
⟨*data*⟩ | ⟨*procedure*⟩

The client area is initially completely blank. After the user has selected or created a project (using the *Project* menu), the client area displays forms for displaying and manipulating box parts. The user should be able to view at least two box parts at a time in the client area. (Designers should refer to [10, chpt. 11] for details on the *multiple-document interface.*)

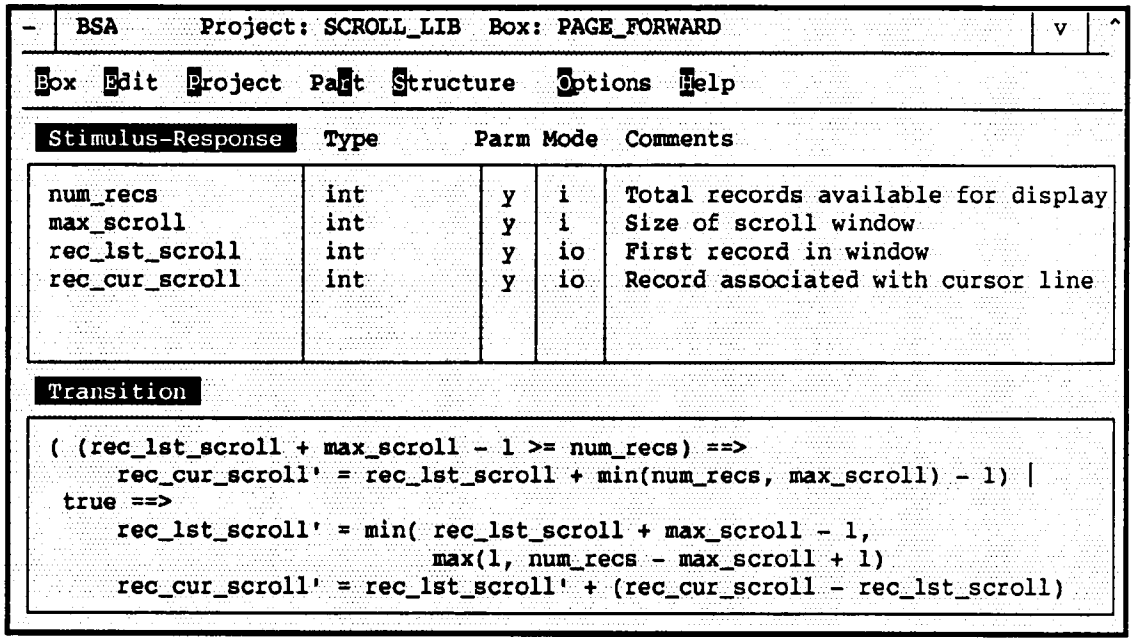
⟨blank⟩ client area:



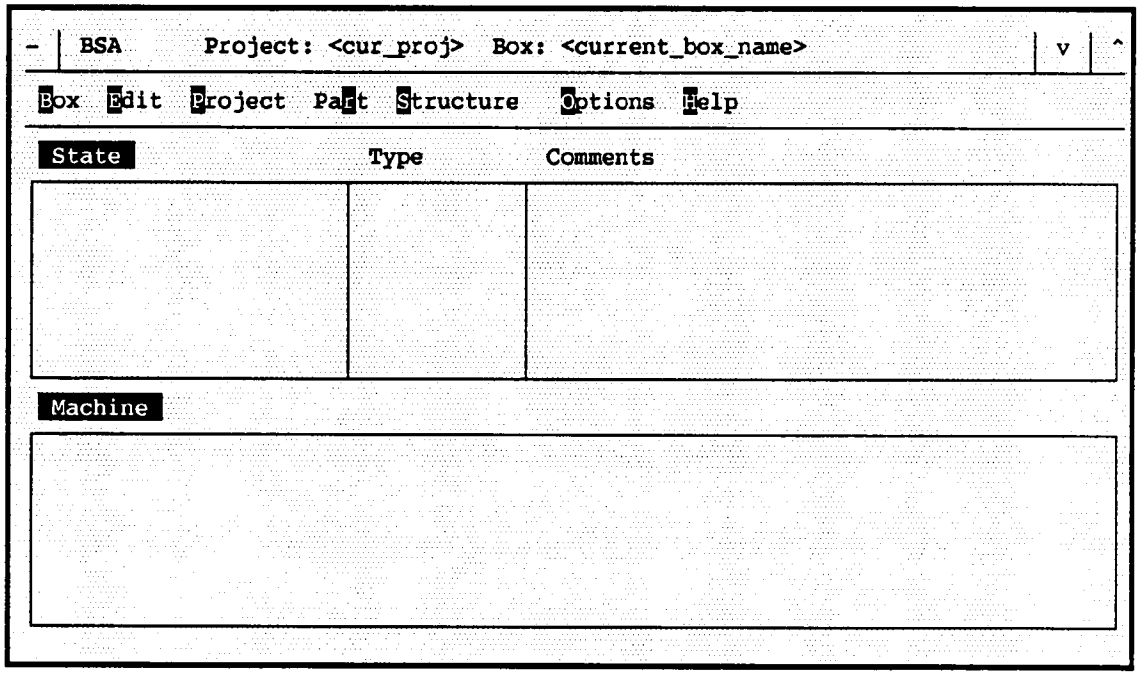
Black box ⟨stimulus_response⟩ and ⟨transition⟩:



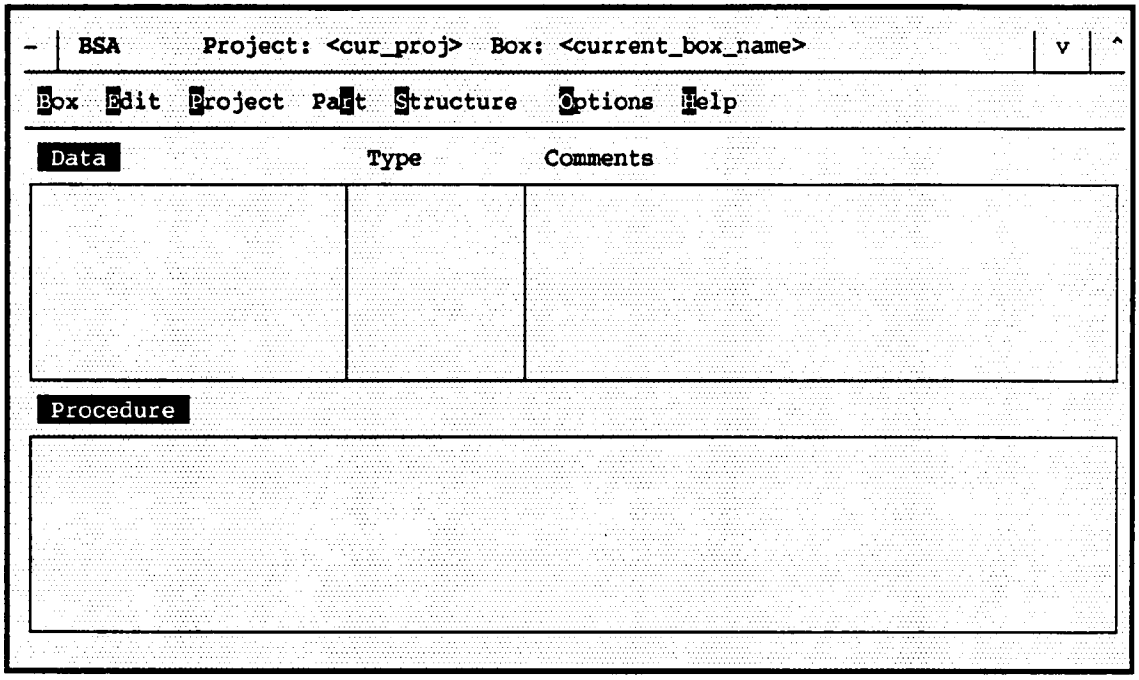
Example of filled-in black box:



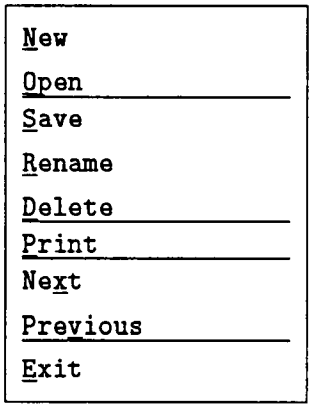
State box ⟨state⟩ and ⟨machine⟩:



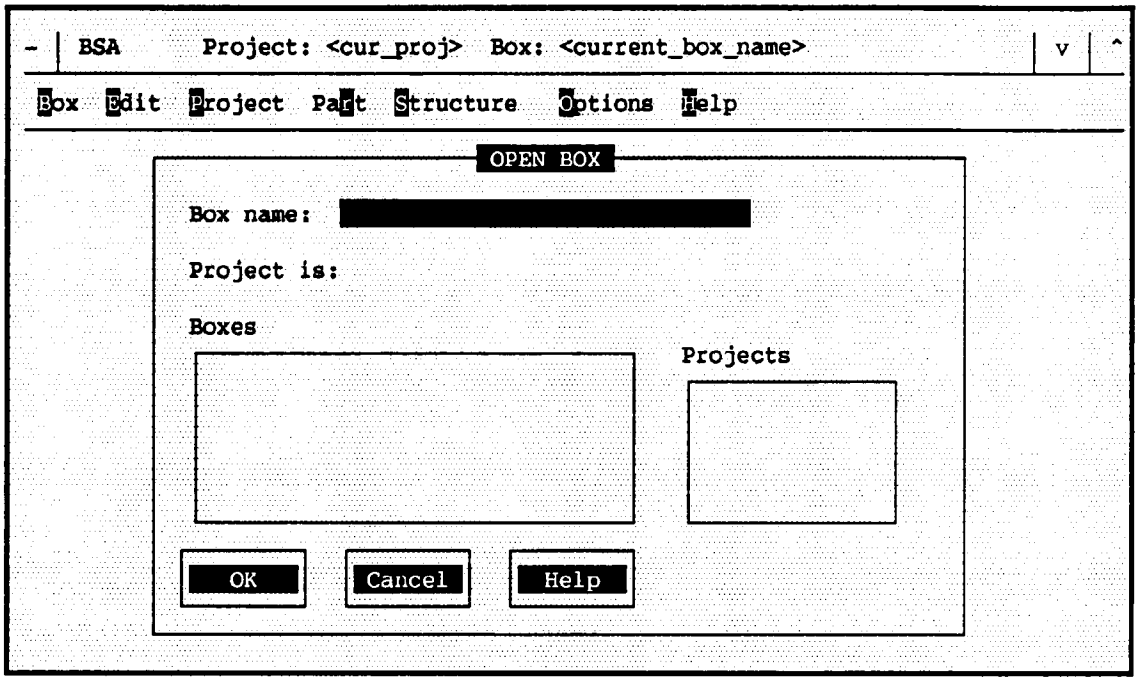
Clear box ⟨data⟩ and ⟨procedure⟩:



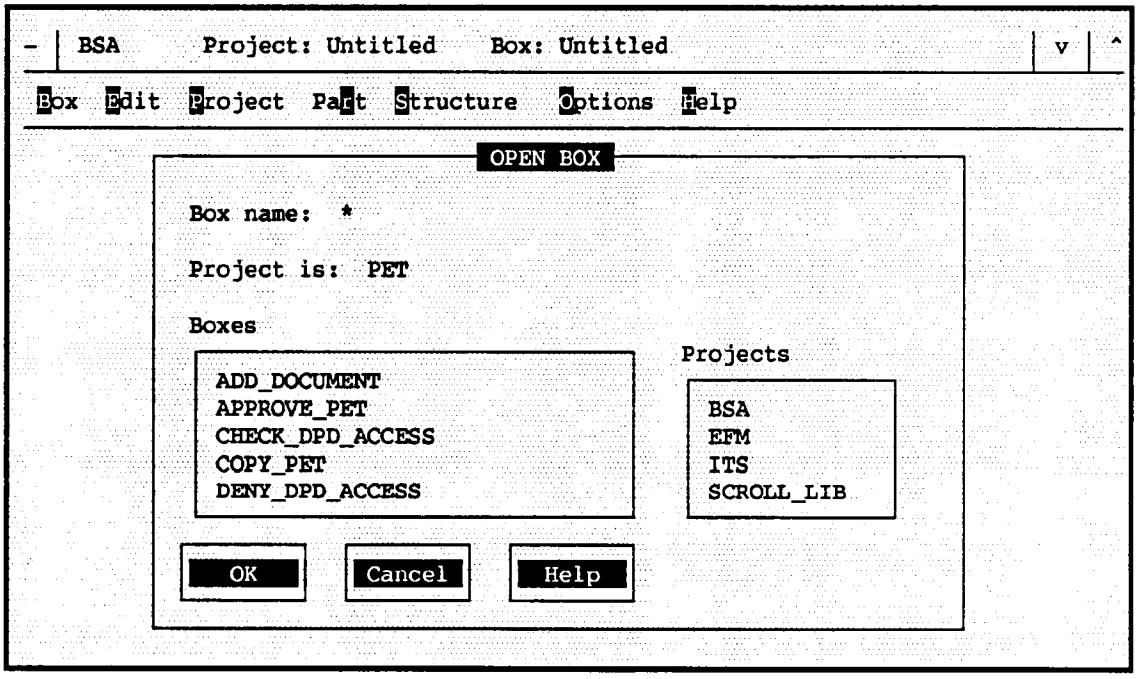
Box_Menu The box menu lists commands which allow the user to select actions that manipulate *boxes* as whole units.



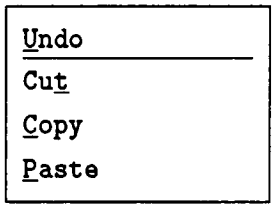
Open_Box Dialogue Panel:



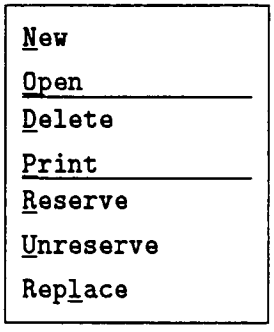
Example of filled in Open_Box:



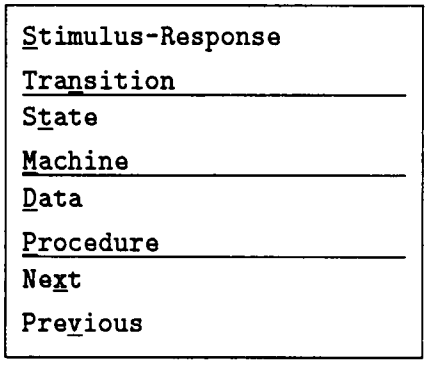
Edit_Menu This menu provides *cut*, *copy*, *paste* and *undo* commands to allow the user to reorganize a design document.



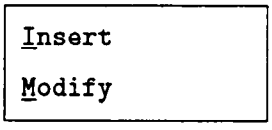
Project_Menu This menu allows the user to select actions to create or modify *projects*. It also provides some limited project librarian functions.



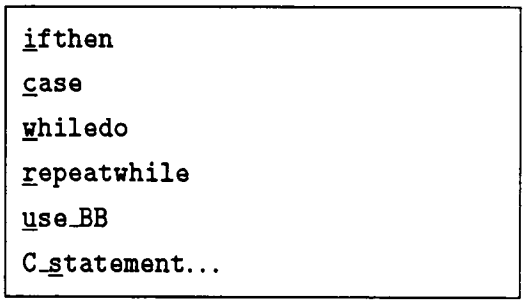
Part_Menu The part menu lists commands that allow the user to reposition the cursor to different parts within the box.



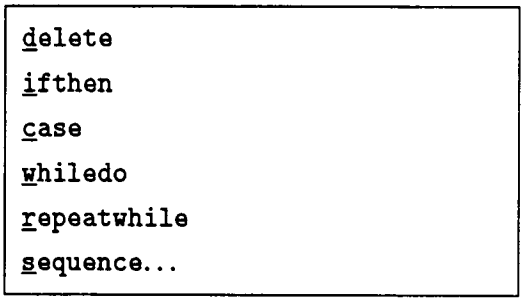
Structure_Menu The structure menu allows the user to insert or modify syntax structures.



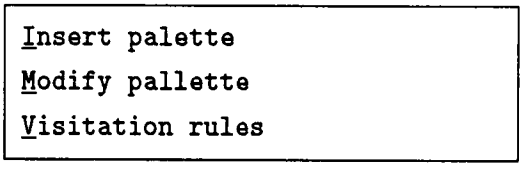
Insert The insert submenu allows the user to select syntax structures to be inserted at the current cursor position.



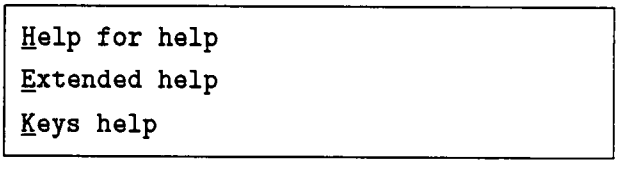
Modify The modify submenu lists commands that can be used to modify syntax structures at the current cursor position. The availability of these commands are dependent on the location of the user's selection cursor.



Option_Menu Menu to allow the user to customize the application. Options provided are *Modify Structures Palette*, *Insert Structures Palette*, and *Visitation Rules.*



Help_Menu Menu to allow the user to request various kinds of help about the BSA application. Options provided are *Help for help...*, *Extended help...*, *and Keys help....*



Data_Item_Form Generic form for entering stimulus-response, state data and machine data lists. Appears in the client window of the primary window.

Text_Editor Generic form for entering text editing objects such as transition rules and data descriptions.

Item_Menu Generic form containing a list of items from which the user may position the light bar to select a desired item. The action pull-down menus are instances of item menus. This kind of menu is also used for data type selection and for selecting projects, boxes, box parts, or syntax structures.

Structure_Editor Generic form that appears in the client area of the primary window to allow viewing and syntax editing of a clear box procedure.

Exit_Form Form displayed just before exiting the BSA application.

8.4.2 Printed Output

The printed output from the Box Structure Assistant includes printing a "pretty-printed" CDL for one box or a group of boxes (See *CDL Box*), and printing a box usage hierarchy indentation chart (see *Usage Hierarchy*).

Design A Box Design Language representation is printed according to the following BNF rewrite rules. The BNF rules also represent the indentation used within each construct.

⟨*CDL_design*⟩ ::=

⟨*CDL_Box*⟩...

⟨*CDL_Box*⟩ ::=

⟨*unit_id*⟩
⟨*CDL_Black_box*⟩
⟨*form_Feed*⟩
⟨*unit_id*⟩
⟨*CDL_State_box*⟩
⟨*form_feed*⟩
⟨*unit_id*⟩
⟨*CDL_Clear_Box*⟩

⟨*CDL_Black_box*⟩ ::=

define_BB ⟨*declaration_specifiers*⟩ ⟨*box_name*⟩
as
[⟨*tab*⟩ **input**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
[⟨*tab*⟩ **inout**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
[⟨*tab*⟩ **output**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
⟨*tab*⟩ **transition**
⟨*tab*⟩ ⟨*tab*⟩ {*transition_rules*} ...
end_BB

⟨*CDL_State_Box*⟩ ::=

define_SB ⟨*declaration_specifiers*⟩ ⟨*box_name*⟩
as
[⟨*tab*⟩ **input**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
[⟨*tab*⟩ **inout**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
[⟨*tab*⟩ **output**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
⟨*tab*⟩ **state**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
⟨*tab*⟩ **transition**
⟨*tab*⟩ ⟨*tab*⟩ {*transition_rules*} ...
end_SB

⟨*CDL_Clear_box*⟩ ::=

define_CB ⟨*declaration_specifiers*⟩ ⟨*box_name*⟩
as
[⟨*tab*⟩ **input**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
[⟨*tab*⟩ **inout**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
[⟨*tab*⟩ **output**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
⟨*tab*⟩ **state**
⟨*tab*⟩ ⟨*tab*⟩ {*parameter_list*} ...
⟨*tab*⟩ ⟨*tab*⟩ {*non_parameter_list*} ...]
⟨*tab*⟩ **proc**
⟨*tab*⟩ ⟨*tab*⟩ ⟨*temp_data_list*⟩
⟨*tab*⟩ ⟨*tab*⟩ ⟨*control_statement_list*⟩
end_CB

Usage Hierarchy A project hierarchy chart is printed according to the following BNF rewrite rules. The BNF rules also represent the indentation used within each construct.

⟨*Usage_Hierarchy*⟩ ::=

⟨*Project_Identification*⟩
⟨*top_box_name*⟩
{ ⟨*tabs*⟩ ⟨*box_name*⟩ }
⋮

⟨*Project_Identification*⟩ ::=

`Usage Hierarchy Chart`
`[Project | Reuse Library]` : ⟨*project_name*⟩
⟨*current_date*⟩
⟨*new_line*⟩
⟨*new_line*⟩

8.4.3 User Messages

This subsection summarizes the various information, warning and action messages that may be provided to BSA users. The values listed in this section are of data type *MESSAGE.*

already_at $\hat{=}$ "You are already at the"[*first* | *last*] ⟨*obj_type*⟩

A generic set of messages that are issued when the user attempts a move in the left or right direction when they are already at the first or last object, respectively.

already_in $\hat{=}$ ⟨*item*⟩ "is already in" ⟨*list_name*⟩

The user attempted to add an item to a list when the item was already in the list.

box_already_known $\hat{=}$ ⟨*box*⟩ " already exists in project library"

box_already_reserved $\hat{=}$ ⟨*box*⟩ " is already reserved"

The user attempted to reserve a box that was already reserved.

box_deleted $\hat{=}$ ⟨*box*⟩ " deleted"

boxes_deleted $\hat{=}$ ⟨*box_deleted*⟩[*boxes_deleted*]

The user has entered the ⟨*Delete_Box*⟩ command to delete a box and all eligible descendents of that box.

box_locked $\hat{=}$ "Box is in use by another user."

The user attempted a modify operation on a box that was being used by another user.

box_name_change $\hat{=}$ ⟨*old_box*⟩ " has been renamed to " ⟨*new_box*⟩

A box has successfully been renamed. All modules that referenced the old box are also changed.

box_not_known $\hat{=}$ ⟨*box*⟩ " not found in project library"

box_not_open $\hat{=}$ "No box has been opened; you must first open or create a box to perform that operation"

The user attempted to perform an operation on a box or box part when no box is currently open.

box_not_reserved $\hat{=}$ ⟨*box*⟩ " is not reserved"

box_read_only $\hat{=}$ "Invalid operation on a read-only box"

An operation was attempted on a box to which the user only has read access.

box_reserved ≙ ⟨*box*⟩ " is reserved by " ⟨*user*⟩

The user reserved a box or attempted an operation that is invalid on a reserved box.

box_replaced ≙ ⟨*box*⟩ " replaced in " ⟨*proj*⟩ " library"

The user replaced a box in the library.

box_unreserved ≙ ⟨*box*⟩ " no longer reserved by " ⟨*user*⟩

box_saved ≙ ⟨*box*⟩ " saved in " ⟨*proj*⟩ " library"

delete_confirmation ≙ ⟨*proj_name*⟩ " deleted"

field_is_full ≙ "field is full"

The user attempted to add a character to a field that was already at it's maximum length.

file_not_found ≙ ⟨*filename*⟩ " not found"

The user attempted an operation on a non-existent file.

new_box_confirm ≙ ⟨*box_name*⟩ " created"

The user has created the definition to a new box.

new_project_confirm ≙ ⟨*proj_name*⟩ "created"

The user has successfully created a new project.

no_field_help_avail ≙ "No help is available for this field"

no_window_help_avail ≙ "No help is available for this window"

ok This is a special value in *MESSAGE* to represent a success condition. No message is actually displayed.

only_in_proc ≙ "That function is only valid in the clear box procedure"

The user attempted an operation which is only valid in the clear box procedure part.

open_project_confirm ≙ ⟨*proj_name*⟩ " selected"

The user successfully opened a project that is known to the system and is not already open.

proj_boxes_in_use ≙ "Boxes in this project are currently reserved or locked by other users."

The user attempted to delete a project when some of the boxes in the project were reserved by other users.

project_already_known ≙ ⟨*proj_name*⟩ "is not a valid project"

The user attempted an operation on a project that is not known to the system.

project_already_open $\hat{=}$ ⟨*proj_name*⟩ " is already open"

The user attempted an operation on a project that was already open.

project_not_known $\hat{=}$ ⟨*proj_name*⟩ " not found"

The user attempted an operation on a project that is not known to the system.

string_not_found $\hat{=}$ ⟨*search_str*⟩ "not found"

8.5. transition

In this section we will give a transition rule for every identified stimulus-response pair. We will use a mixture of informal and formal notations.

In $\mathcal{V}$ predicates, for example, *stim_hist* is used to reference the stimulus history of BSA and *mrs* refers to the most recent stimulus accepted by BSA for the current user. $\mathcal{R}$ predicates may make assertions about response variables which are suffixed by !. For example, the assertion *user_message*! = *at_first_box* means that the message "You are already at the first box" will be displayed to the user in the ⟨*user_message*⟩ area of the ⟨*primary_window*⟩. When formal assertions are given in the $\mathcal{V}$ or $\mathcal{R}$ sections, they will be followed by informal explanations if needed. We will assume multiple lines that are listed in $\mathcal{V}$ or $\mathcal{R}$ predicates are separated by logical *and-connectives*($\wedge$) unless another connective is present.

1. Begin Session

1.1 *Invoke_System*(*user* : $USERNAME$)

If possible, the identification of the user should be obtained from the operating system.

$\mathcal{V}1$: *stim_hist* = ⟨⟩ $\vee$ *mrs* = *Exit_Session*
This predicate identifies the set of stimulus histories for beginning a user session with BSA. Either the stimulus history is null or the most recent stimulus was an *Exit_Session*.

$\mathcal{R}1$: Display *Welcome_Form* for 3 seconds or until the user presses the *OK* button. Then display *Primary_Window* with a blank *client_area* and a current box name of "`Untitled`."
The last project opened or created will be displayed as the current project unless it was subsequently deleted. If no project has been opened or created then the current project will be named "`Untitled`." *current_box*! = "`Untitled`"
client_area! = *blank*

$\mathcal{V}2$: *stim_hist* $\neq$ ⟨⟩ $\wedge$ *mrs* $\neq$ *Exit*

$\mathcal{R}2$: [to be determined] Note: one possibility for this condition is that the user was trying to begin a new session just after a power failure or some other abnormal end to the previous session. The integrity of the system should be tested. If the system state is valid, release all locks owned by the current user and allow the user to continue. Otherwise, report a fatal error and exit.

1.2 *Action_Select*

V1: The user may invoke the *Action_Select* menu anytime the primary window is active. This would be true when *mrs* = *Invoke_System* and when the user is not interacting with a sub-menu dialogue.

R1: The action menu bar menu shows the following selections:
⟨*Box* | *Edit* | *Project* | *Part* | *Structure* | *Options* | *Help*⟩

2. Box Operations

2.1 *Box*

V1: *mrs* = *action_select* and a current project has been selected.

R1: Display the box menu choices: ⟨*New* | *Open* | *Save* | *Rename* | *Delete* | *Print* | *Next_box* | *Prev_box* | *Exit*⟩

2.2 *New_box (box : BOX_NAME, proj : PROJ_ABBR)*

The default value for *proj* is the *current_project.* Ordinarily, a box is created automatically when a user creates a reference to a new box when inserting a *use BB* statement for a non-existent box.

V1: *box* is not a known box within the project *proj*

R1: If a current box exists, the user is prompted to save any changes made to the current box. The specified *box* becomes the current box and the first part of the box is displayed in the *client_area* with the cursor positioned to the first entry field in that part.
current_project! = *proj*
current_box! = *box*

V2: The stimulus history contains a ⟨*New_box*⟩ command that created the box. It was not followed by a subsequent command to delete or rename the box.

R2: *user_message!* = *box_already_exists*

2.3 *Open_box(box : BOX_NAME, proj : PROJ_ABBR)*

The user will be able to select the box name from a pick list of all boxes for the specified project or he may enter the box name directly. The current project is the default project name, but the user may select a project from a pick list of known projects or type in a project name.

V1: *box* is a known box within the project *proj*
box is not currently reserved or locked
The phrase "*box* is a known box" means that it was previously created (by a ⟨*New_box*⟩ or ⟨*Create_Box_Def*⟩ command) for the specified project (the current project is the default) and was not subsequently deleted (by a ⟨*Delete_box*⟩ command).
A box is *reserved* if the stimulus history contains a *Reserve_Box* command for the box not followed by a *Replace_Box* or *Unreserve_Box* command.

R1: *current_project!* = *proj*
current_box! = *box*
The *client_area* is updated to show the first part of the previously saved box. The cursor is positioned to the first entry field within the first box part.

$\mathcal{V}2$: *box* is not known within the project *proj*
$\mathcal{R}2$: *user_message!* = *box_not_known*

$\mathcal{V}3$: *box* is currently reserved
$\mathcal{R}3$: *user_message!* = *box_reserved*

2.4 *Save_Box*

$\mathcal{V}1$: *Modify access was granted for the current box*
$\mathcal{R}1$: *user_message!* = *box_saved*
current_project' = *current_project*
current_box' = *current_box*
client_area! = *client_area*
This function saves all editing changes for the current box that were made since the box was accessed. A user message is displayed to indicate the box has been saved. The other information on the display remains unchanged.

$\mathcal{V}2$: *Modify access was not granted for the current box*
$\mathcal{R}2$: *user_message!* = *box_read_only*

2.5 *Rename (old_box : BOX_NAME, new_box : BOX_Name)*

Prompt user for the old box name (the default is the current box name). If the box exists within current project then prompt user for new box name, validate that new box name does not already exists, and rename the box (and all references to the box).

$\mathcal{V}1$: *old_box* = *current_box*
new_box is not known within *current_project*
$\mathcal{R}1$: *current_box!* = *new_box*
user_message! = *box_name_change*
The existing box is renamed to the new box name. All references to the old box are also changed to the new box name.

$\mathcal{V}2$: *old_box* $\neq$ *current_box*
old_box is known within *current_project*
new_box is not known within *current_project*
old_box is not on reserve
$\mathcal{R}2$: *user_message!* = *box_name_change*
The existing box is renamed to the new box name. All references to the old box are also changed to the new box name. The response here is the same as $\mathcal{R}1$ except that the *current_box* does not change.

$\mathcal{V}3$: *old_box* is on reserve
$\mathcal{R}3$: *user_message!* = *box_reserved*

$\mathcal{V}4$: *old_box* is not known within *current_project*
$\mathcal{R}4$: *user_message!* = *box_not_known*

$\mathcal{V}5$: *new_box* is known within *current_project*
$\mathcal{R}5$: *user_message!* = *box_already_exists*

2.6 *Delete_Box (box : BOX_NAME, proj : PROJ_ABBR)*

$\mathcal{V}1$: *box* = *current_box*

box and the descendent boxes of *box* are eligible for deletion

$\mathcal{R}1$: *current_box*! ="Untitled"

user_message! = *boxes_deleted*

If a box is a member of the current project library and is only used by the specified box or by other descendents of the specified box then the descendent box is eligible for deletion. List box to be deleted along with all eligible descendent boxes for deletion confirmation. If the user confirms, then delete the specified box and all eligible descendent boxes.

$\mathcal{V}2$: *box* $\neq$ *current_box*

all descendent boxes of *box* are eligible for deletion

box is a known box in project *proj*

$\mathcal{R}2$: *current_box*! = *current_box*

This response is the same as $\mathcal{R}1$ except that the current box remains the same.

$\mathcal{V}3$: *box* is not a known box in project *proj*

$\mathcal{R}3$: *user_message*! = *box_not_known*

2.7 *Print_Box (box: BOX_NAME, dev: DEV_TYPE, fmt : FORMAT)*

$\mathcal{V}1$: *box* is known within *current_project*

$\mathcal{R}1$: Format the selected box according to the ⟨*CDL_BOX*⟩ syntax for the specified output device. Transmit the formatted file to the specified device. If the device is to display then allow the user to view the box. (i.e., Display the box provide commands to scroll forwards and backwards through the file in units of one line, one screen full or to the beginning or end of the file).

$\mathcal{V}2$: *box* is not known within *current_project*

$\mathcal{R}2$: *user_message*! = *box_not_known*

2.8 *Next_box*

$\mathcal{V}1$: A current box has been selected that is not the last box in the visitation sequence. The visitation sequence is the order in which boxes are opened or created during a session.

$\mathcal{R}1$: Prompt the user to save any changes made to the current box. Move the cursor to the beginning of the next box. The next box is opened and becomes the current box.

$\mathcal{V}2$: The current box is the last box in the visitation sequence.

$\mathcal{R}2$: *user_message! = at_last_box*

$\mathcal{V}3$: The user has not selected a current box.

$\mathcal{R}3$: The ⟨*Next_box*⟩ command is unavailable.

2.9 *Prev_box*

V1: There is a previous box to visit (according to the user specified box visitation rules).

R1: Prompt the user to save any changes made to the current box. Move the cursor to the beginning of the previous box. The previous box is opened and becomes the current box.

V2: The current box is the first box in the visitation order.

R2: *user_message! = at_first_box*

V3: The user has not selected a current box.

R3: The ⟨*Prev_box*⟩ command is unavailable.

2.10 *Exit_Session*

V1: *true*

R1: **if** changes made to current box
then prompt user to Save_Box
fi
Display *Exit_form*
Exit BSA

3. Edit Menu Commands

3.1 *Edit*

V1: *mrs = Action_Select*

R1: Display edit menu choices: ⟨*Undo* | *Cut* | *Copy* | *Paste*⟩

3.2 *Undo*

V1: The most recently executed edit action was a ⟨*Cut*⟩

R1: The system is returned to the same state that it was in prior to executing the ⟨*Cut*⟩ command. ⟨*Undo cut*⟩ re-inserts the text that was cut, selects the text, and the contents of the clipboard are restored to what it contained prior to the ⟨*Cut*⟩.

V2: The most recently executed edit action was a ⟨*Paste*⟩

R2: The system is returned to the same state that it was in prior to executing the ⟨*Paste*⟩ command. ⟨*Undo paste*⟩ removes the text that was pasted and formats the box part so it appears as it was prior to the paste. The selection state is restored. The contents of the clipboard remain unchanged.

V3: The most recently executed edit action was a *Insert Construct* command (see the structure menu commands).

R3: The system is returned to the same state that is was in prior to executing the *Insert Construct*

V4: The most recently executed edit action was a *Modify Construct* command (see the structure menu commands).

R4: The system is returned to the same state that is was in prior to executing the *Modify Construct*

V5: The most recently executed edit action was not *Cut, Paste, Insert Construct, or Modify Construct.*

R5: The undo command is unavailable to the user.

3.3 *Cut*

V1: The user has selected a sequence of text or structure data that has not been subsequently de-selected.

R1: The selected sequence is copied to the clipboard and then removed from the object being edited. The space occupied by the removed text is compressed.

V2: Nothing has been selected to be cut from the object being edited.

R2: null response. In this case the ⟨*cut*⟩ command should not be available for the user to invoke.

3.4 *Copy*

V1: The user has selected a sequence of text or structures from the object being edited that has not been subsequently de-selected.

R1: The selected sequence is copied to the clipboard without removing it from the object being edited.

V2: Nothing has been selected.

R2: The ⟨*Copy*⟩ command is unavailable to the user.

3.5 *Paste*

V1: The user has cut or copied a sequence of text to the clipboard and the clipboard sequence is compatible [to be determined] within the current context.

R1: Copy the contents of the clipboard into the object being edited at the current cursor location. If the insertion is a structure which contains placeholders, the cursor is moved to the first placeholder, otherwise the cursor is moved to the first position following the insertion.

4. Project Operations

4.1 *Project*

V1: *mrs* = *Action_Select*

R1: Display Project Menu Choices: ⟨*New*⟩ | ⟨*Open*⟩ | ⟨*Delete*⟩ | ⟨*Print*⟩ | ⟨*Reserve*⟩ | ⟨*Unreserve*⟩ | ⟨*Replace*⟩.

4.2 *New_Project (proj : PROJ_ABBR, reuse : BOOL)*

V1: *proj* is not known

R1: *user_message*! = *new_project_confirm*
current_project! = *proj*
current_box! = "`Untitled`" *client_area*! = *blank*

V2: *proj* is known

R2: *user_message*! = *project_already_known*

The phrase "*proj* is not known" means that *mrs* = *Action_Select* and *proj* must not have been previously created (by the *New_Project* command) unless it was subsequently deleted (by the *Delete_Project* command). *New_Project* allows a user

to create a new project. Certain projects will be designated as reuse projects. A *reuse* project may contain boxes which may be used by boxes from other projects.

4.3 *Open_Project (proj : PROJ_ABBR)*

$\mathcal{V}1$: *proj* is known to the system
proj is not currently open

$\mathcal{R}1$: *user_message*! = *open_project_confirm*
current_project! = *proj*
current_box! = "Untitled"
client_area! = *blank*
The selected project (*proj*?) becomes the current project. The *client_area* will be blank until the user has selected a box.
The phrase "*proj* is known" means it was previously created by a *New_Project* command but not subsequently deleted by a *Delete_Project* command. A project is "not currently open" if the most recent *New_Project*($p1, r$) or *Open_Project*($p1$) command was not for the same project (i.e., $p1 \neq proj$).

$\mathcal{V}2$: *proj* is already open

$\mathcal{R}2$: *user_message*! = *project_already_open*

$\mathcal{V}3$: *proj* is not known

$\mathcal{R}3$: *user_message*! = *project_not_known*

4.4 *Delete_Project (proj : PROJ_ABBR)*

$\mathcal{V}1$: *proj* is known $\wedge$*proj* is not open
No boxes in *proj* are locked

$\mathcal{R}1$: *user_message*! = *delete_confirmation*

$\mathcal{V}2$: *proj* is known $\wedge$*proj* is open.

$\mathcal{R}2$: *user_message*! = *delete_confirmation*
client_area! = *blank* (remove project from display)
return to untitled project
current_project! = "Untitled"
current_box! = "Untitled"

$\mathcal{V}3$: *proj* is not known

$\mathcal{R}3$: *user_message*! = *project_not_known*

$\mathcal{V}4$: boxes in *proj* are locked or reserved

$\mathcal{R}4$: *user_message*! = *proj_boxes_inuse*

4.5 *Print_Hierarchy (device:DEV_TYPE, fmt:FORMAT)*

$\mathcal{V}1$: A current project has been selected. This means that within the current session, the user must have selected a current project. A current project is the most recent project selected by either the ⟨*New*⟩ or ⟨*Open*⟩ command, (but has not been deselected by a subsequent ⟨*Delete*⟩ command).

$\mathcal{R}1$: Format a usage hierarchy indentation chart for the current project. Transmit the formatted chart to the specified output media. If output media is display, then allow user to view the result using the box display commands.

4.6 *Reserve_Box (box : BOX_NAME, proj : PROJ_ABBR, user: USERNAME)*

V1: *box* is a known box
box is not already reserved

R1: *user_message!* = *box_reserved*
This action reserves the box in the user's name and creates a file containing the reserved box in the user's current directory.

V2: *box* is a known box
box is reserved

R2: *user_message!* = *box_already_reserved*

V3: *box* is not a known box

R3: *user_message!* = *box_not_known*

4.7 *Unreserve_Box (box : BOX_NAME, proj : PROJ_ABBR, user : USERNAME)*

V1: *box* is a known box
box is reserved by *user*

R1: *user_message!* = *box_unreserved*

V2: *box* is a known box
box is reserved by different user.

R2: *user_message!* = *box_reserved*

V3: *box* is a known box
box is not reserved

R3: *user_message!* = *box_not_reserved*

V4: *box* is not a known box

R4: *user_message!* = *box_not_known*

4.8 *Replace_Box (box : BOX_NAME, proj : PROJ_ABBR, user : USERNAME)*

V1: *box* is a known box
box is reserved by *user*
A file with same name as *box* name with `.CDL` extension exists in the user's current directory.
The file is valid (contents are in the expected format and connections with other boxes have same arguments and types)

R1: *user_message!* = *box_replaced*

V2: *box* is a known box
box is reserved by *user*
A file named ⟨*box*⟩`.CDL` exists in user's current directory
File is not valid

R2: A list of error messages listing unexpected or invalid expressions found in the reserved file.

V3: *box* is a known box
box is reserved by *user*
A file named ⟨*box*⟩.CDL is not found in user's current directory.
R3: *user_message!* = *file_not_found*

V4: *box* is a known box
box is reserved by a different user
R4: *user_message!* = *box_reserved*

V5: *box* is a known box
box is not reserved
R5: *user_message!* = *box_not_reserved*

V6: *box* is not known in the project
R6: *user_message!* = *box_not_known*

5. Part Operations

This submenu allows the user to choose the next box *part* to be worked by using 'item menu' commands *(Prev_Item, Next_Item, Select).* We use the variable *current_part* to indicate which box part is currently being displayed in the client area. For example, *current_part* = *stimulus_response* means the black box stimulus-response part is being displayed. If no current box is selected then *current_part* = *null.* The values for this variable are drawn from the set *BOX_PART* which contains the following values:

$$BOX\_PART = \{stimulus\_response,\ transition,\ state,\ machine,\ data,\ procedure,\ null\}$$

The order for visiting and displaying these box parts is described by the following *next_part* function.

$$next\_part = \{(stimulus - response, transition), (transition, state), (state, machine), (machine, data), (data, procedure), (procedure, stimulus - response)\}$$

5.1 *Part*

V1: *mrs* = *action_select* and a current box has been selected.
R1: Display the part menu choices: ⟨*Next_Part* | *Prev_Part* | *Stimulus_response* | *Transition* | *State* | *Machine* | *Data* | *Procedure*⟩.

5.2 *Next_Part*

V1: *current_part* $\neq$ *null*
A current box has been selected.
R1: *current_part'* = *next_part*(*current_part*)
Moves the cursor to the next part in the design part sequence.
V2: *current_part* = *null*
No current box has been selected
R2: *User_message!* = *box_not_open*

5.3 *Prev_Part*

$\mathcal{V}1$: $current\_part \neq null$

A current box has been selected

$\mathcal{R}1$: $current\_part' = next\_part^{-1}(current\_part)$

We use the relational inverse operator which has the effect of reversing the ordered pairs of the *next_part* function. This operation moves the cursor to the previous part in the design part sequence.

$\mathcal{V}2$: $current\_part = null$

No current box has been selected

$\mathcal{R}2$: $User\_message! = box\_not\_open$

5.4 *Stimulus_response*

$\mathcal{V}1$: $current\_part \neq null$

A current box has been selected.

$\mathcal{R}1$: $current\_part' = stimulus\_response$

Move to the black box stimulus-response part

$\mathcal{V}2$: $current\_part = null$

$\mathcal{R}2$: $user\_message! = box\_not\_open$

5.5 *Transition*

$\mathcal{V}1$: $current\_part \neq null$

A current box has been selected.

$\mathcal{R}1$: $current\_part' = transition$

Move to the black box transition part.

$\mathcal{V}2$: $current\_part = null$

$\mathcal{R}2$: $user\_message! = box\_not\_open$

5.6 *State*

$\mathcal{V}1$: $current\_part \neq null$

A current box has been selected.

$\mathcal{R}1$: $current\_part' = state$

Move to the state box data part.

$\mathcal{V}2$: $current\_part = null$

$\mathcal{R}2$: $user\_message! = box\_not\_open$

5.7 *Machine*

$\mathcal{V}1$: $current\_part \neq null$

A current box has been selected.

$\mathcal{R}1$: $current\_part' = machine$

Move to the state box machine part.

$\mathcal{V}2$: $current\_part = null$

$\mathcal{R}2$: $user\_message! = box\_not\_open$

5.8 *Data*

$\mathcal{V}1$: *current_part* $\neq$ *null*
A current box has been selected.

$\mathcal{R}1$: *current_part'* = *data*
Move to the clear box machine data part.

$\mathcal{V}2$: *current_part* = *null*

$\mathcal{R}2$: *user_message!* = *box_not_open*

5.9 *Procedure*

$\mathcal{V}1$: *current_part* $\neq$ *null*
A current box has been selected.

$\mathcal{R}1$: *current_part'* = *procedure*
Move to the clear box procedure part.

$\mathcal{V}2$: *current_part* = *null*

$\mathcal{R}2$: *user_message!* = *box_not_open*

6. Structure Menu Operations

The structure menu operations are available for inserting or modifying CDL syntax structures when the user is editing the CDL clear box procedure.

6.1 *Structure*

$\mathcal{V}1$: *current_part* = *procedure*
The user is working on a clear box procedure part.
The user has modify access to the box.

$\mathcal{R}1$: Display the Structure_Menu. The choices on this menu are $\langle Insert \mid Modify \rangle$.
The user may delete a construct using the edit menu's *Cut* command.

$\mathcal{V}2$: *current_part* $\neq$ *procedure*
The clear box procedure is not shown in the client area.

$\mathcal{R}2$: *user_message!* = *only_in_proc*

$\mathcal{V}3$: The user doesn't have modify access to the current box.

$\mathcal{R}3$: *user_message!* = *box_read_only*

6.2 *Insert*

An accelerator key should be provided to directly invoke this submenu.

$\mathcal{V}1$: *current_part* = *procedure*
A clear box procedure part is shown in the client area.

$\mathcal{R}1$: Display an item menu listing CDL syntax structures. This menu lists the following syntax structures: $\langle ifthen \rangle$, $\langle case \rangle$, $\langle whiledo \rangle$, $\langle repeatwhile \rangle$, $\langle use\_BB \rangle$, and $\langle C\_statement \rangle$.

$\mathcal{V}2$: *current_part* $\neq$ *procedure*
The clear box procedure is not shown in the client area.

$\mathcal{R}2$: *user_message!* = *only_in_proc*

6.3 *use_BB(box : BOX_NAME)*

$\mathcal{V}1$: *box* is a known box in *current_project* $\vee$
box is a known box in a reusable project library.

$\mathcal{R}1$: A dialogue box appears with a template for entering the box parameters. The sequence and data types of the arguments are preset according to the called box. Default variable names will appear if names and types match between the arguments of the used box and existing variables of the using box. The user can only enter or modify the variable names for this usage. Any variable names entered must be known variables in the stimulus-response, state, or data parts of the current box. Also, their data types must correspond to the calling parameters of the used box.

$\mathcal{V}2$: *box* is not a known box in *current_project*
box is not a known box in a reusable project library

$\mathcal{R}2$: A dialogue box appears to allow the user to create the interface definition to the new box. (See *Create_box_def* below.) The user enters a list of calling arguments with variable names, data types, and in/out mode indicators. This has the effect of creating and saving a new box for the current project.

6.4 *User selects a syntax structure for insertion*

$\mathcal{V}1$: $mrs = \langle Insert \rangle \wedge current\_part = procedure$

$\mathcal{R}1$: A corresponding structure template is inserted in front of the user's current statement. The cursor is then moved to the first non-terminal placeholder within the inserted template.

6.5 *Create_Box_Def(box : BOX_NAME, args : ARG_LIST)*

This command is invoked when the user enters a *use_BB* to a box that is undefined either in the current project or any project marked as a reuse project.

$\mathcal{V}1$: *box* is not a known box in the current project
box is not a known box in a reuse project

$\mathcal{R}1$: $user\_message! = new\_box\_confirm$

6.6 *Modify*

An accelerator key should be provided to directly invoke this function.

$\mathcal{V}1$: $current\_part = procedure$
A clear box procedure part is shown in the client area.

$\mathcal{R}1$: Display an item menu listing commands that may be used to modify the syntax structure at the current cursor position. For example, if the cursor is positioned over the $\langle if \rangle$ token of an $\langle ifthen \rangle$ statement, the possible commands would include: $\langle delete \rangle$, $\langle sequence \rangle$, $\langle case \rangle$, $\langle whiledo \rangle$, and $\langle repeatwhile \rangle$.

$\mathcal{V}2$: $current\_part \neq procedure$
The clear box procedure is not shown in the client area.

$\mathcal{R}2$: $user\_message! = only\_in\_proc$

6.7 *User selects a syntax modification structure*

$\mathcal{V}1$: *mrs* = *Modify* $\wedge$ *current_part* = *procedure*

$\mathcal{R}1$: Display the modified structure. (Note: at this point the user must be able to use the edit menu's *undo* command to return the screen to the state it was in before the modification.) As an example, the ⟨*sequence*⟩ command would convert the *ifthen* structure to a sequence by removing the outer syntax tokens (⟨*if*⟩, ⟨*then*⟩, and ⟨*fi*⟩) and the conditional expression of the *ifthen* statement but, would leave the embedded statements within the ⟨*then*⟩ and ⟨*else*⟩ clauses. The ⟨*delete*⟩ command would remove the entire structure between the ⟨*if*⟩ and matching ⟨*fi*⟩ tokens. The ⟨*case*⟩, ⟨*whiledo*⟩, or ⟨*repeatwhile*⟩ command would convert the *ifthen* structure into a *case*, *whiledo*, or *repeatwhile* statement, respectively.

7. Options Menu

The options menu is intended to allow the user to customize the application in various ways.

7.1 *Options*

$\mathcal{V}1$: *current_part* = *procedure*

$\mathcal{R}1$: Display the option menu consisting of the following options: ⟨*New Structures Palette, Modify Structures Palette, and Visitation Rules*⟩.

$\mathcal{V}2$: *current_part* $\neq$ *procedure*

The clear box procedure is not shown in the client area.

$\mathcal{R}2$: *user_message*! = *only_in_proc*

7.2 *New Structures Palette*

This option allows the user to choose whether to keep a display on the screen listing the syntax structure choices for insertion. This display only appears while the user is editing a clear box procedure.

$\mathcal{V}1$: *current_part* = *procedure*

$\mathcal{R}1$: Display a dialogue box that would allow the user to choose whether the palette of new syntax structures should remain accessible at all times.

$\mathcal{V}2$: *current_part* $\neq$ *procedure*

The clear box procedure is not shown in the client area.

$\mathcal{R}2$: *user_message*! = *only_in_proc*

7.3 *Modify Structures Palette*

This option allows the user to keep a display on the screen listing all possible commands for modifying the syntax structure at the current users position.

$\mathcal{V}1$: *current_part* = *procedure*

$\mathcal{R}1$: Display a dialogue box that would allow the user to choose whether the palette of syntax modification commands should remain accessible at all times.

$\mathcal{V}2$: *current_part* $\neq$ *procedure*

The clear box procedure is not shown in the client area.

$\mathcal{R}2$: *user_message*! = *only_in_proc*

7.4 *Visitation Rules*

V1: *current_part* = *procedure*

R1: Invoke a dialogue box to allow the user to specify options for box visitation order.

V2: *current_part* $\neq$ *procedure*

The clear box procedure is not shown in the client area.

R2: *user_message*! = *only_in_proc*

8. Help Menu

The user should be provided with the following options for requesting help at any point in the application.

8.1 *Context Help*

This command provides specific information about the item on which the selection cursor is positioned. Users invoke this function by pressing the ⟨*F1*⟩ function key or by selecting the ⟨*Help*⟩ pushbutton.

V1: Context help is defined for the user's current cursor position.

R1: Display contextual help information about the item on which the selection cursor is positioned.

V2: Context help is not defined for the user's current cursor position.

R2: *user_message*! = *no_field_help_avail*

8.2 *Help*

V1: *true*

R1: Display the *Help_Menu* which displays the following options: ⟨*Help for help, Extended help, Keys help*⟩.

8.3 *Extended Help*

Command to request information about the tasks that can be performed in the current application window.

V1: Extended help information is available for the current window.

R1: Display information about what tasks they can perform in the window from which they requested *Extended Help.*

V2: No extended help information is available for the current window.

R2: *user_message*! = *no_window_help_avail*

8.4 *Help Help*

V1: *true*

R1: Display information about how to use the help facility.

8.5 *Keys Help*

V1: *true*

R1: Display information describing the key assignments of the BSA application.

9. Data Item Form Operations

These commands are available for editing and navigation within a *Data_Item_Form.* The box parts of *stimulus-response*, *state*, and *data* fit into this category.

To simplify the definition of these commands, we will model each *field* as a pair of finite sequences of characters *left, right*, such that *left* $\frown$ *right* is the complete field value. A *record* consists of a fixed size sequence of fields. The data item form itself consists of a variable size sequence of records. Pairs of sequences are used to model the user's cursor position in a field or in the sequence of records. In each case, the left element of the pair represents objects (records or characters) prior to the user's current cursor position and the right element of the pair represents objects on or after the current cursor position. The head of the right pair represents the current position. The set of characters which may appear in a field will be denoted as CH. The specification methods applied below were adapted from [26].

9.1 $Insert\_ch(ch : CH; left, right : \mathrm{seq}(CH))$
ch is the character to be inserted, *left* is the sequence of characters that precede the text cursor, *right* represents the sequence of characters after the cursor position, with the cursor considered to be in between *left* and *right*.

V1: *true*

R1: $left' = left \frown ch$
$right' = right$
For example, if *Insert_ch(*"p","**Exam**","**le**"*)*,
then after the operation $left'$ ="**Examp**" and
$right'$ ="**le**".

9.2 $Overstrike\_ch(ch : CH; left, right : \mathrm{seq}(CH))$

V1: *true*

R1: An overstrike character operation may be similarly defined:
$left' = left \frown ch$
$right' = tail(right)$

9.3 *Ins_Ovr_Toggle*

V1: The *Ins_Ovr_Toggle* command has been pressed an odd number of times during this user session.

R1: Future character key strokes invoke the *Overstrike_ch* command. The cursor changes to a thin shape.

V2: *true*

R2: Future character key strokes will invoke the *Insert_ch* command. The cursor changes to a thick shape.

9.4 *Move (obj_type : OBJ_TYPE, dir: DIRECTION)*
This family of operations allows the user to move the cursor in either direction from one character, field or record to the next one.

V1: Another object of type *obj_type* exists in the current direction and the object at the current location is *acceptable*.

R1: Move to the next object.

V2: The object at the current location is not acceptable.

R2: Display appropriate error message.
The cursor location does not change.

$\mathcal{V}3$: There are no more object of type *obj_type* in the current direction.
$\mathcal{R}3$: *user_message*! = already at [*first* | *last*] ⟨*obj_type*⟩
The cursor location does not change.

9.5 $Del\_left\_ch(left, right : \mathrm{seq}\, CH)$
$\mathcal{V}1$: $left \neq \langle\rangle$
$\mathcal{R}1$: $left' = front(left)$
$right' = right$
$\mathcal{V}2$: $left = \langle\rangle$
$\mathcal{R}2$: *nullresponse*

9.6 $Del\_right\_ch(left, right : \mathrm{seq}\, CH)$
$\mathcal{V}1$: $right \neq \langle\rangle$
$\mathcal{R}1$: $left' = left$
$right' = tail(right)$
$\mathcal{V}2$: $left = \langle\rangle$
$\mathcal{R}2$: *nullresponse*

9.7 $Blank\_field(left, right : \mathrm{seq}\, CH)$
This operation blanks out the current field.
$\mathcal{V}1$: *true*
$\mathcal{R}1$: $left' = \langle\rangle$
$right' = \langle\rangle$

9.8 $Del\_rec(left\_recs, right\_recs : \mathrm{seq}\, RECORD)$
This operation deletes the current record. The current record is the *head* record in the sequence *right_recs*.
$\mathcal{V}1$: *true*
$\mathcal{R}1$: $right\_recs' = tail(right\_recs)$
$left\_recs' = left\_recs$
$cursor\_field' = 1$
$cursor\_char' = 1$
The cursor is repositioned to the first character position in the first field of the record following the deleted record. We introduce the variables *cursor_field* and *cursor_char* to indicate the location of the cursor within new current record.

9.9 $Open\_left\_rec(left\_recs, right\_recs : \mathrm{seq}\, RECORD)$
Command to open a record slot before the current record.
$\mathcal{V}1$: *true*
$\mathcal{R}1$: $right\_recs' = blank\_rec \frown right\_recs$
$left\_recs' = left\_recs$
$cursor\_field' = 1$
$cursor\_char' = 1$
The variable *blank_rec* is introduced to represent a record containing blank values for all fields in the record.

9.10 *Pick_item*

$\mathcal{V}1$: *true*

$\mathcal{R}1$: This user operation is designed for entering a value into the DATA_TYPE field of a DATA_ITEM_FORM. The user selects the data_type of a particular data item by selecting from a predefined list of data types which are presented in an ITEM menu. See the transition rules for the ITEM menu below. The selected value from the ITEM menu is inserted into the form. The user is allowed to enter a new data type if needed.

10. Text Editor Operations

Text editor have some of the same operations as the *data item form*.

10.1 *Insert_ch (ch : CH; left, right* seq *CH)*

$\mathcal{V}1$: *true*

$\mathcal{R}1$: $left' = left \frown ch$

$right' = right$

10.2 *Overstrike_ch (ch : CH; left, right* seq *CH)*

$\mathcal{V}1$: *true*

$\mathcal{R}1$: $left' = left \frown ch$

$right' = tail(right)$

10.3 *Ins_Ovr_Toggle*

$\mathcal{V}1$: The *Ins_Ovr_Toggle* command has been pressed an odd number of times during this user session.

$\mathcal{R}1$: Future character key strokes invoke the *Overstrike_ch* command. The cursor changes to a thin shape.

$\mathcal{V}2$: *true*

$\mathcal{R}2$: Future character key strokes will invoke the *Insert_ch* command. The cursor changes to a thick shape.

10.4 *Move (obj : OBJ_TYPE, dir : DIRECTION)*

$\mathcal{V}1$: *true*

$\mathcal{R}1$: See similar *data item form* command.

10.5 *Delete (obj : OBJ_TYPE, dir : DIRECTIONS)*

$\mathcal{V}1$: *true*

$\mathcal{R}1$: See similar *data item form* commands.

11. Item Menu Operations

An item menu is used for selecting from a small number of finite choices. The term *selection cursor* refers to a highlighted option in an item menu.

11.1 *Prev_Item*

$\mathcal{V}1$: The selection cursor is not on the first item.

$\mathcal{R}1$: Move the selection cursor to the previous item.

$\mathcal{V}2$: The selection cursor is on the first item.

$\mathcal{R}2$: Move the selection cursor to the last item.

11.2 *Next_Item*

$\mathcal{V}1$: The selection cursor is not on the last item.

$\mathcal{R}1$: Move the selection cursor to the next item.

$\mathcal{V}2$: The selection cursor is on the last item.

$\mathcal{R}2$: Move the selection cursor to the first item.

11.3 *Mneumonic(ch : CH)*

This cursor positioning command is available where a unique *mneumonic* character is assigned to each menu choice. The *mneumonic* character for a choice is always identified by being underlined.

$\mathcal{V}1$: *ch* is a mneumonic for a choice in the item menu.

$\mathcal{R}1$: Move the selection cursor to the corresponding choice.

$\mathcal{V}2$: *ch* is not a mneumonic for a choice in the item menu.

$\mathcal{R}2$: The selection cursor does not move.

11.4 *Abbreviate(ch : CH)*

"In selection fields that contain a list of choices for which unique mnemonics cannot be assigned, users can type a letter to move the selection cursor to the next choice starting with that letter."[10, page 28] This technique assumes that this kind of item list always in alphabetical order.

$\mathcal{V}1$: There exists a choice in the item menu whose first character is *ch*.

$\mathcal{R}1$: The selection cursor moves to the first choice in the item menu whose first character is *ch*.

$\mathcal{V}2$: *true*

$\mathcal{R}2$: The selection cursor does not move.

11.5 *Select*

$\mathcal{V}1$: *true*

$\mathcal{R}1$: Select the currently highlighted item.

11.6 *Add_Item(item :* seq *(CH),*
list : seq (seq *(CH)))*

For some item menus, a function may be provided in the item menu to allow the user to add an item to the list.

$\mathcal{V}1$: *item* is not already in this item menu.

$\mathcal{R}1$: Insert *item* in the item menu so that item menu remains in alphabetical order.

$\mathcal{V}2$: *true*

$\mathcal{R}2$: *user_message*! = *already_in*

12. Box Display Operations

12.1 *Move (obj : OBJ_TYPE, dir : DIRECTION)*

$\mathcal{V}1$: Another object of type *obj* exists in the *dir* direction.

$\mathcal{R}1$: Move the cursor to the beginning of the next object in the *dir* direction.

$\mathcal{V}2$: *true*

$\mathcal{R}2$: *user_message*! = *already_at*

12.2 *Search (search_str : TEXT, dir : DIRECTION)*

$\mathcal{V}1$: *search_str* is found in the current box between the current cursor position and the left or right end of the box depending on the direction of the search.

$\mathcal{R}1$: Move cursor to beginning of the matching string.

$\mathcal{V}2$: *true*

$\mathcal{R}2$: *user_message*! = *string_not_found*

CHAPTER 9

The BSA State Box Specification

This chapter presents the BSA state box specification which consists of a *state* section and a *machine* section. The *state* section presents abstract state data to encapsulate the stimulus histories of the black box. State data will be represented using the Z notation. Invariant relationships among these data and their initial states will also be presented. The *machine* section will present the operations of the state box.

9.1. state

In the subsections that follow we will 1) list data types to be used in the Z schemas, 2) define mathematical tools that will be used to describe aspects of the state data and operation schemas, 3) define the state data and invariant conditions, and 4) declare the initial state of the system.

9.1.1 Data types

The following list defines all data types used in the state box specification. Basic types are enclosed in brackets ([ ]). As a convention, data types are shown in all upper case letters. Schema names are shown with the initial letters of each word in the schema name in upper case. Variable names are given in all lower case.

$ACCESS ::= read \mid modify$ This type contains values that are used to control user access to a box or box part.

$BOX\_NAME == IDENT$ Identifiers that can be used to name boxes. *BOX_NAME* is introduced to make the specification more readable, but both *BOX_NAME* and *IDENT* refer to the same data type.

$BOX\_ID == BOX\_NAME \times PROJ\_ABBR$

This set is used as short hand for specifying a box in a particular project.

BOX_PART A fixed set used to identify the components of a box.

$$BOX\_PART ::= stimulus\_response \mid transition \mid state \mid machine \mid data \mid procedure \mid null$$

BUTTON ::= *yes* | *no* | *cancel* | *OK* | *save* Describes the user's selection of a dialogue push button.

[CH] Identifies the set of characters that may be entered into a CDL document (see *TEXT*).

[DEV_TYPE] A set of possible media on which information may be output (disk file, printer, display).

DIRECTION ::= *left* | *right* A set of values used to indicate the direction of movement relative to the cursor.

DISPLAY The set of display forms that can appear on the user's screen. The values for this type were defined in the **Display Forms** subsection (8.4.1) of the black box response specification.

ED_CMD ::= *cut* | *paste* | *copy* | *nil* Type of edit actions.

ED_TYPE ::= *text* | *data* | *proc* | *nil* Indicates the kind of information being edited.

[FORMAT] Format for printing a design (e.g., TEX or standard).

[IDENT] Set of possible identifiers that can be used to name boxes or data variables (see *BOX_NAME*).

INS_OVR ::= *insert* | *overstrike* This data type is used to indicate whether keyed characters will replace existing characters or will instead be inserted into the text.

MESSAGE Messages that may be displayed to the user. The values for this type were defined in the **User Messages** subsection of the black box response specification.

MODE ::=| *in* | *out* | *inout* Set of values to indicate whether a parameter is a stimulus, a response, or both.

[PRED] Basic type for the predicates used within conditional statements.

PROC Clear box procedure. This type is defined using the following *free type* definition:

$$
\begin{aligned}
PROC ::= {} & nilP \mid \\
& statementP \mid \\
& usebbP \mid \\
& sequenceP\langle\!\langle \mathrm{seq}\, PROC \rangle\!\rangle \mid \\
& ifthenP\langle\!\langle PRED \times PROC \times PROC \rangle\!\rangle \mid \\
& caseP\langle\!\langle PRED \times \mathrm{seq}\,(PRED \times PROC) \times PROC \rangle\!\rangle \mid \\
& whiledoP\langle\!\langle PRED \times PROC \rangle\!\rangle \mid \\
& dountilP\langle\!\langle PROC \times PRED \rangle\!\rangle
\end{aligned}
$$

The Z *free type* definition allows us to describe recursive structures such as lists and trees and is equivalent to the following axiom definition:

$[PROC]$

$$
\begin{array}{|l}
nilP : PROC \\
statementP : PROC \\
usebbP : PROC \\
sequenceP : \mathrm{seq}\, PROC \overset{inj}{\rightarrow} PROC \\
ifthenP : PRED \times PROC \times PROC \overset{inj}{\rightarrow} PROC \\
caseP : PRED \times \mathrm{seq}\,(PRED \times PROC) \times PROC \overset{inj}{\rightarrow} PROC \\
whiledoP : PRED \times PROC \overset{inj}{\rightarrow} PROC \\
dountilP : PROC \times PRED \overset{inj}{\rightarrow} PROC \\
\hline
\mathbf{disjoint}\langle \{statementP\}, \{usebbP\}, \\
\quad \mathrm{rng}\, sequenceP, \mathrm{rng}\, ifthenP, \mathrm{rng}\, caseP, \\
\quad \mathrm{rng}\, whiledoP, \mathrm{rng}\, dountilP \rangle \\
\\
\forall \mathrm{W} : \mathbb{P}\, PROC \bullet \\
\quad \{statementP, usebbP\} \cup sequenceP(\!|\mathrm{seq\ W}|\!) \\
\qquad \cup\, ifthenP(\!|PRED \times \mathrm{W} \times \mathrm{W}|\!) \\
\qquad \cup\, caseP(\!|PRED \times \mathrm{seq}\,(PRED \times \mathrm{W}) \times \mathrm{W}|\!) \\
\qquad \cup\, whiledoP(\!|PRED \times \mathrm{W}|\!) \\
\qquad \cup\, dountilP(\!|\mathrm{W} \times PRED|\!) \subseteq \mathrm{W} \\
\qquad \Rightarrow PROC \subseteq \mathrm{W}
\end{array}
$$

In this definition, *statementP* and *usebbP* are called *constants* and the remaining declarations are called *constructors.* The predicate lists two axioms constraining the constants and constructors. First, all the constants are distinct, and any constructor will not produce a constant or a result that could be obtained from a different constructor. The second axiom says that the smallest subset of *PROC* which contains all the constants and is closed under the constructors is *PROC* itself.

[PROJ_ABBR] Set of valid project abbreviations that are used to identify projects.

$REUSE\_IND ::= yes \mid no$ Used to indicate whether a project will serve as a reuse library.

$TEXT == \mathrm{seq}\, CH$ Text is considered a sequence of characters.

[TYPE] Contains the set of CDL data types that may be assigned to variable identifiers.

[USERNAME] Contains names to identify users.

9.1.2 Mathematical Tool-Kit Additions

By using generic and axiomatic descriptions, we can extend the standard Z tool-kit (as listed in the appendix) to provide specialized tools for describing our system. In this section we define some higher-order functions which we use later in defining state invariants and state transitions. The Z *generic* constant construct allows us to define these functions over a generic (unknown) data type. Generic constants are defined in a box that has a double bar across the top with formal generic type parameters. The generic type parameters are usually instantiated implicitly from the type of the expression.

The *nodup* function is used to convert a sequence into an injective sequence. It works by removing ordered pairs from a sequence if the range element has already been mapped by a lower domain element.

$$[X]$$

$$nodup : \operatorname{seq} X \rightarrow \operatorname{iseq} X$$

$$\forall s : \operatorname{seq} X \bullet$$
$$nodup\ S = \{m : \operatorname{dom} S \mid (\forall n : s^{-1}(\!|s(m)|\!) \bullet m \leq n)\} \overset{seq}{\lhd} s$$

For example, $nodup\langle T, E, N, N, E, S, S, E, E\rangle = \langle T, E, N, S\rangle$.

The function *strip* provides a method to convert a sequence of order pairs into a sequence of single elements.

$$[X,\ Y]$$

$$strip : \operatorname{seq}(X \times Y) \rightarrow \operatorname{seq} X$$

$$\forall s : \operatorname{seq}(X \times Y) \bullet$$
$$strip(s) = \lambda n : \operatorname{dom} s \bullet first\ s(n)$$

For example, $strip\langle(C, X), (A, Y), (T, Z)\rangle = \langle C, A, T\rangle$.

To model some of the edit menu commands we define the relation $\hookrightarrow$ ("suffixed by"), and introduce the function *truncate*.

$$[X]$$
$$\hookrightarrow : \operatorname{seq} X \leftrightarrow \operatorname{seq} X$$
$$truncate : (\operatorname{seq} X \times \operatorname{seq} X) \nrightarrow \operatorname{seq} X$$

$$\forall s, t : \operatorname{seq} X \bullet$$
$$t \hookrightarrow s \Leftrightarrow \exists r : \operatorname{seq} X \bullet r \frown s = t$$
$$\forall r, s : \operatorname{seq} X \bullet truncate(r \frown s, s) = r$$

To illustrate, if $t = \langle C, L, E, A, N, R, O, O, M\rangle$ and $s = \langle R, O, O, M\rangle$ then $t \hookrightarrow s = true$ and $truncate(t, s) = \langle C, L, E, A, N\rangle$.

Given a sequence of characters and a set of special delimiter characters to find, the distance to the next character following a delimiter is given by *find_nt*.

$$[X]$$

$$find\_nt : (\mathbb{P}\, X \times \operatorname{seq} X) \nrightarrow \mathbb{N}$$

$$\forall \beta : \mathbb{P}\, X; \mathcal{S} : \operatorname{seq} X \mid (\exists k : \beta \bullet k \in \operatorname{rng} \mathcal{S}) \bullet$$
$$find\_nt(\beta, \mathcal{S}) = min\{i : 2 \mathinner{..} \#\mathcal{S} \mid \mathcal{S}(i-1) \in \beta \wedge \mathcal{S}(i) \notin \beta\}$$

For example, $find\_nt(\{sp, nl\}, \langle A, sp, nl, T, a, k, e\rangle) = 4$.

Given a sequence of characters and a set of special delimiter characters to find, the distance to the last delimiter which preceeds a non-delimiter is given by *find_ld*.

$$[X]$$

$$find\_ld : (\mathbb{P}\, X \times \operatorname{seq} X) \nrightarrow \mathbb{N}$$

$$\forall \beta : \mathbb{P}\, X; \mathcal{S} : \operatorname{seq} X \mid (\exists k : \beta \bullet k \in \operatorname{rng} \mathcal{S}) \bullet$$
$$find\_ld(\beta, \mathcal{S}) = max\{i : 1 \mathinner{..} \#\mathcal{S} - 1 \mid \mathcal{S}(i) \notin \beta \wedge \mathcal{S}(i+1) \in \beta\}$$

For example, $find\_ld(\{sp, nl\}, \langle nl, H, O, W, sp, nl, S, O\rangle) = 6$.

The generic function *find_first* is used to compute the index of the first occurrance of a pattern with in a sequence and *find_last* computes the index of the last matching pattern in the sequence.

$$[X]$$

$$find\_first : (\operatorname{seq} X \times \operatorname{seq} X) \nrightarrow \mathbb{N}$$
$$find\_last : (\operatorname{seq} X \times \operatorname{seq} X) \nrightarrow \mathbb{N}$$
$$\overset{\cdots}{\hookrightarrow} : \operatorname{seq} X \leftrightarrow \operatorname{seq} X$$

$$\forall s, t : \operatorname{seq} CH \bullet$$
$$(t \overset{\cdots}{\hookrightarrow} s \Leftrightarrow (\exists r, r' : \operatorname{seq} X \bullet r \frown s \frown r')$$
$$t \overset{\cdots}{\hookrightarrow} s \Rightarrow$$
$$(find\_next(s,t) = min\{i : 0 \mathinner{..} (\#t - \#s) \mid (succ^i \fatsemi t) \overset{\cdots}{\hookrightarrow} s\} \wedge$$
$$find\_last(s,t) = max\{i : 0 \mathinner{..} (\#t - \#s) \mid (succ^i \fatsemi t) \overset{\cdots}{\hookrightarrow} s\})$$

To illustrate,

$$find\_first(\langle bad\rangle, \langle abadabado\rangle) = 2$$
$$find\_last(\langle bad\rangle, \langle abadabado\rangle) = 6$$

The following schema is used to return the index of the first item in a list of items which begins with a specified letter.

$[X]$

$$first\_match : (X \times \text{seq}(\text{seq}\, X)) \pfun \mathbb{N}$$

$$\forall \alpha : X; \mathcal{L} : \text{seq}(\text{seq}\, X) \mid \alpha \in (\text{rng}\, \mathcal{L})(\!|1|\!) \bullet$$
$$first\_match(\alpha, \mathcal{L}) = min\{i : 1 \,..\, \#\mathcal{L} \mid \mathcal{L}(i)(1) = \alpha\}$$

Suppose we have a sequence of character sequences $L = \langle\langle C, A, T\rangle, \langle D, O, G\rangle, \langle K, I, D\rangle\rangle$. Then $first\_match(\text{“D”}, L) = 2$.

We use the schema *Sort_List* for sorting injective sequences into sorted order. The input and output of *Sort_List* are injective sequences of generic type X which has a total order “$<_x$” defined upon it. A total order $<_x$ is defined upon X if and only if for all $x, y, z : X$ the following are true:

$$\neg(x <_x x) \land$$
$$((x <_x y) \lor (x = y) \lor (y <_x x)) \land$$
$$((x <_x y) \land (y <_x z) \Rightarrow (x <_x z))$$

Sort_List[X]

$$in?, out! : \text{iseq}\, X$$

$$\forall i, j : \text{dom}\, out! \bullet$$
$$(i < j) \Rightarrow (out?(i) <_x out?(j))$$
$$\text{rng}\, in? = \text{rng}\, out!$$

9.1.3 State Schemas

From the black box description of BSA, we readily see the need to represent *boxes*, *projects*, and *parts*. We begin by representing the contents of a box.

Box_Contents

$$box\_name : IDENT$$
$$proj\_abbr : PROJ\_ABBR$$
$$interface, stim\_resp, state, data : \text{iseq}\, IDENT$$
$$transition, machine : TEXT$$
$$procedure : PROC$$

$$box\_data, parameters : \mathbb{P}\, IDENT$$
$$data\_access\_mode : IDENT \pfun MODE$$
$$data\_type : IDENT \pfun TYPE$$

$$data\_comments : IDENT \nrightarrow TEXT$$

$$box\_calls : \text{seq}(BOX\_ID \times \text{seq}(IDENT))$$
$$boxes\_used : \text{seq}(BOX\_ID)$$

These declarations describe the parts of a box. The variable *box_name* gives the name of the box and *proj_abbr* names the project library where the box resides. The sequence of calling arguments that are needed to *use* this box are listed in *interface*. The black box, state box and clear box data are represented as *injective* sequences of identifiers in the variables *stim_resp*, *state*, and *data*, respectively. An *injective* sequence is a sequence which contains no repetitions. The black box *transition* rules and the state box *machine* parts are of data type *TEXT*. The clear box *procedure* uses the data type *PROC*.

The set *box_data* contains all variables defined within the box unit. The set *parameters* contains the set of variables that are marked as parameters in the stimulus-response list. The function *data_access_mode* gives the mode (*in*, *out*, or *inout*) of each identifier. Data types of each variable are given by the *data_type* function. The function *data_comments* maps identifiers to their descriptive comments. The *box_calls* sequence represents the sequence of lower level boxes that are invoked from this box. The *boxes_used* injective sequence lists the sequence of boxes used but eliminates the repetitions.

The next step is to describe static or invariant properties of the data within *Box_Contents* using the schema *Box*.

Box

Box_Contents

$$\langle \text{rng } stim\_resp, \text{rng } state, \text{rng } data \rangle \textbf{ partitions } box\_data$$

$$box\_name \in box\_data \Rightarrow$$
$$(box\_name \in \text{rng } stim\_resp \wedge box\_name \in parameters \wedge$$
$$data\_access\_mode(box\_name) = out)$$

$$interface = stim\_resp \overset{seq}{\rhd} parameters$$
$$\text{rng } interface = parameters$$

$$\text{rng } stim\_resp \subseteq \text{dom } data\_access\_mode$$
$$\text{dom } data\_access\_mode \subseteq box\_data$$
$$\text{dom } data\_type = box\_data$$
$$\text{dom } data\_comments \subseteq box\_data$$

$$boxes\_used = nodup(strip(box\_calls))$$
$$\text{rng rng rng } box\_calls \subseteq box\_data$$

The **partitions** predicate states the property that variables declared in *stim_resp, state,* and *data* are distinct and span the set *box_data.* According to the second predicate, the *box_name* may be declared as a response parameter. The third predicate gives the relationship between the box interface and the stimulus-response list. The *interface* to the box is derived from the *stim_resp* list by filtering out any non-*parameters.*

The fifth predicate says that every stimulus-response variable must have a declared data access mode. We leave open the possibility that other variables in the box could have a declared access mode, but any variable with an access mode must be in *box_data.* Every variable in the box must have a declared data type. As an option, variables in the box may have descriptive comments associated with them.

We make use of the *nodup* and *strip* functions to show that *boxes_used* is derived from *box_calls* by stripping the argument lists and eliminating the duplicates. The final predicate says that the identifiers used in the argument lists of *box_calls* must be declared in *box_data.*

Boxes are organized into projects as described by the following schema:

Projects

$$known\_projects, reuse\_lib, proj\_lib : \mathbb{P}\ PROJ\_ABBR$$
$$repository : \mathbb{P}\ Box$$
$$idBox : BOX\_ID \overset{inj}{\nrightarrow} Box$$
$$box\_proj : BOX\_NAME \leftrightarrow PROJ\_ABBR$$
$$box\_reuse : BOX\_NAME \twoheadrightarrow PROJ\_ABBR$$

$$reuse\_lib \cup proj\_lib = known\_projects$$
$$reuse\_lib \cap proj\_lib = \emptyset$$
$$idBox = \{b : repository \bullet (b.box\_name, b.proj\_abbr) \mapsto b\}$$
$$\text{rng}\,(\text{dom}\ idBox) \subseteq known\_projects$$
$$\text{rng}\ box\_proj \subseteq proj\_lib$$
$$\text{rng}\ box\_reuse \subseteq reuse\_lib$$
$$box\_proj \cup box\_reuse = \text{dom}\ idBox$$

The set *known_projects* captures the black box notion of a project that has been created but not deleted. The sets *reuse_lib* and *proj_lib* represent two kinds of projects. A *reuse_lib* project contains boxes that may be used by other projects. The boxes in a *proj_lib* project are used only by that project. The second declaration represents the collection of all boxes in the system. The injective function *idBox* guarantees that no two boxes in the system will have the same box identifier. The *box_proj* relation lists all boxes that

are associated with project libraries. Note that *box_proj* could be defined equivalently as $box\_proj : \mathbb{P}\ BOX\_ID$ since *BOX_ID* is an abbreviation for $BOX\_NAME \times PROJ\_ABBR$. The *box_reuse* function contains box identifiers of boxes associated with reuse libaries. By defining *box_reuse* as a function, all boxes in reusable libraries must have distinct names. The first two predicates state that every known project can be categorized as either a project library or a reuse library. The third predicate defines *idBox* as a function that is derived directly from the *repository* of boxes. The fourth predicate limits our repository to boxes associated with *known_projects*. It is permissible, however, to have projects that do not have any boxes associated with them. Every box in *box_proj* is associated with a project library and every box in *box_reuse* is associated with a reuse library. The final invariant says that every box in the repository is identified in either *box_proj* or *box_reuse*.

The librarian commands provided by the project menu require us to recognize boxes that have been *reserved* but not yet *unreserved* or *replaced* by another user. To support cooperative project work, we also must provide a locking mechanism to prevent more than one user from updating a box at a time.

Box_Categories

$$known\_boxes, available, locked, reserved : \mathbb{P}\ BOX\_ID$$
$$reserved\_by : BOX\_ID \nrightarrow USER\_ID$$
$$locked\_by : BOX\_ID \nrightarrow USER\_ID$$

$$available \cup locked \cup reserved = known\_boxes$$
$$available \cap (locked \cup reserved) = \emptyset$$
$$locked = \mathrm{dom}\ locked\_by$$
$$reserved = \mathrm{dom}\ reserved\_by$$

The set *known_boxes* contains identifiers of all boxes that have been created but not deleted. Boxes that are not *locked* or *reserved* are in the set *available*. The *locked_by* and *reserved_by* functions records who has a box locked or reserved.

Our system is more than just a collection of boxes. It is structured into a usage hierarchy; linked together by *use_BB* statements. The integrity of these links must be maintained when *use_BB* statements are inserted or modified, when the *stimulus-response* parts of existing boxes are modified, when boxes are deleted, and when reserved boxes are replaced into the repository.

Usage_Hierarchy

$$uses : BOX\_ID \leftrightarrow BOX\_ID$$
$$used\_by : BOX\_ID \leftrightarrow BOX\_ID$$
$$all\_uses : BOX\_ID \leftrightarrow BOX\_ID$$
$$top\_boxes : \mathbb{P}\ BOX\_ID$$

$$uses = used\_by^{-1}$$

$$all\_uses = uses^{+}$$
$$\text{dom } uses \setminus \text{dom } used\_by \subseteq top\_boxes$$
$$\text{dom } used\_by \cap top\_boxes = \emptyset$$

The relations *uses* and *used_by* are inverses of each other and represent the usage hierarchy from opposite points of view. The ordered pairs in the *uses* relation represent boxes that *directly* use other boxes. The *all_uses* is the *transitive closure* of *uses* and describes both direct and indirect usage relationships between pairs. All boxes that use other boxes but are not themselves used by other boxes must exist in the set of *top_boxes*. Also, the set *top_boxes* will not contain any boxes that are *used_by* other boxes.

The functions *next_part* and *prev_part* are used to define the usual order for visiting parts.

$$next\_part, prev\_part : BOX\_PART \overset{inj}{\nrightarrow} BOX\_PART$$

$$next\_part = \{(stimulus\_response, transition),$$
$$(transition, state), (state, machine),$$
$$(machine, data), (data, procedure),$$
$$(procedure, stimulus\_response)\}$$
$$prev\_part = next\_part^{-1}$$

There are certain invariant properties that relate *Projects*, *Usage_Hierarchy*, and *Box_Categories*. The set of *known_boxes* identifies all boxes that are contained in the repository. Also, every box in *known_boxes* is either a top level box or it is used by other boxes. Boxes cannot use other boxes from other project libraries, but they can use boxes from reuse libraries. With these invariant conditions the Box Structure Assistant is partly described by the schema *BSA:*

BSA

$$Projects$$
$$Usage\_Hierarchy$$
$$Box\_Categories$$

$$known\_boxes = \text{dom } idBox$$
$$known\_boxes = top\_boxes \cup \text{dom } used\_by$$
$$\forall bp : uses \mid \#\text{rng } bp > 1 \bullet second(bp) \in box\_reuse$$
$$\forall b : repository \bullet \text{rng } b.used\_boxes = used\_by(\!| b.box\_name \times b.proj\_abbr |\!)$$

The third predicate requires that every module that is used by a project other than its own, must itself be a member of a reuse library. For example, if we let

$$uses = \{((box1, projA), (box2, projA)),$$

$((box1, projA), (box3, projB)),$
$((box2, projA), (box4, projC))\}$ and

$box\_reuse = \{(box3, projB)\}$

then the first member of *uses* (bp_1) is itself a relation:

$bp_1 = \{(box1, projA),$
$(box2, projA)\}$

and $\#\text{rng}\, bp_1 = 1$ which does not meet the first condition. Thus, the element bp_1 is not covered by the quantifier. For the second element bp_2, $\#\text{rng}\, bp_2 = 2$, which meets the first condition and $(box3, projB) \in box\_reuse$, which meets the second condition. The third element also meets the condition imposed by the first predicate but fails to satify the second condition since $(box4, projC) \notin box\_reuse$. Therefore, the system would be in an invalid state that is disallowed by the invariant. The final predicate is used to show the relationship between the *used_boxes* sequence and the *used_by* relation. The Box Structure Assistant allows the user to create new boxes or edit existing boxes, but the changes do not become permanent until the *Save_Box* command is invoked. We define the following *buffer* or temporary work area to represent histories where a *New_Box* or *Open_Box* command has been issued but not yet followed by a *Save_Box* command.

Buffer

$user : USERNAME$
$buffer\_access : ACCESS$
$entry\_mode : INS\_OVR$
$cur\_box : Box$
$current\_part : BOX\_PART$
$current\_box : BOX\_NAME$
$current\_project : PROJ\_ABBR$

$current\_box = cur\_box.box\_name$
$current\_project = cur\_box.proj\_abbr$

Each box a user creates or opens during a session is recorded in a visit sequence. This sequence is used to determine which box will be opened by the *Next_Box* and *Prev_Box* commands.

Visit_Sequence

$visits : \text{iseq}(BOX\_ID)$

$\text{rng}\, visits \in known\_boxes$

Between the last *Exit_Session* command and the next *Invoke_System* command, the

system must remember the last project that was opened by the user. To encapsulate this set of stimulus histories we use a function mapping user names to current projects.

User_Info

$$user\_project : USERNAME \pfun PROJ\_ABBR$$

9.1.4 Initial States

To correctly define a state machine, we must declare the initial state of the system. First we define an initial state of a created *Box* and then use this definition to define the initial state of the system.

Init_Box

$$Box$$

$$box\_data = \emptyset$$
$$transition = \langle\rangle$$
$$machine = \langle\rangle$$
$$procedure = nilP$$

Because of the invariant conditions for *Box*, setting $box\_data = \emptyset$ implicitly initializes other box components to the empty set.

The BSA black box specification considered the condition of a null stimulus history ($stim\_hist = \langle\rangle$). We use the following schema to represent this initial state.

Init_BSA

$$BSA$$

$$known\_projects = \emptyset$$
$$known\_boxes = \emptyset$$

The initial state of *Buffer* when the system stimulus history is null ($stim\_hist = \langle\rangle$) is described by the following schema:

Init_Buffer

$$Buffer$$

$$user = nil$$
$$buffer\_access = read$$
$$current\_project = \text{“}\texttt{Untitled}\text{”}$$
$$current\_box = \text{“}\texttt{Untitled}\text{”}$$

$current\_part = null$

$\theta cur\_box = \theta Init\_Box$

The initial state of the *user_project* function is the empty set.

Init_User_Info

$User\_Info$

$user\_project = \emptyset$

9.2. machine

In this section we will use Z operation schemas to define the state box transition rules. Each state box transition rule has a correponding black box transition rule that describes identical behavior. The state box transitions, however, reference state data as defined in Z state space schemas instead of the stimulus history.

1. Begin Session

1.1 *Invoke_System(usr : USERNAME)*

Our black box transition rule for this stimuli checks the condition $mrs \neq Exit\_Session$. The most recent stimulus command for the current user is not equal to *Exit_Session*. To detect this condition in the state box, we will check to see if the user owns any locks on any boxes. This would only be possible if the user last exited because of a system failure.

Invoke_System

$\Delta Box\_Categories$

$\Delta Buffer$

$\Delta Visit\_Sequence$

$\Xi User\_Info$

$usr? : USERNAME$

$disp\_seq! : \text{seq } DISPLAY$

$usr? \in \text{rng } locked\_by \Rightarrow locked\_by' = locked\_by \rhd \{usr?\}$

$disp\_seq! = \langle welcome\_form, primary\_window \rangle$

$(usr? \in \text{dom } user\_project \Rightarrow current\_project' = user\_project(usr?) \lor$

$usr? \notin \text{dom } user\_project \Rightarrow current\_project' = \text{“}\texttt{Untitled}\text{”})$

$current\_box' = \text{“}\texttt{Untitled}\text{”}$

$current\_part' = null$

$user' = usr?$

$buffer\_access' = read$

$visits' = \langle\rangle$

We adopt a deliberately simple, non-detailed approach to specifying display output. Our main focus is on the computational aspects of the system and at this stage we do not wish to overly constrain possible user interfaces that could be implemented. The sequence *disp_seq!* is used here to indicate that the system response to the *Invoke_System* stimulus is to first display the *welcome_form* followed by the *primary_window.*

The *current_project* for the user is retrieved from the *user_project* function. If this is the user's first session or if no project was open when the user last exited the system, the current_project will be "`Untitled`". The *current_box* for the session is "`Untitled`"and the *current_part* is null.

2. Box Operations

2.1 *Box_Menu*

The following schema simply says that the *Box_Menu* command will display the *box_menu.*

Box_Menu

$disp\_form! : DISPLAY$

$disp\_form! = box\_menu$

2.2 *New_Box (box : BOX_NAME, proj : PROJ_ABBR)*

The *New_Box* command is used to create a new box for a project.

New_Box_OK

$\Xi Box\_Categories$

$\Delta Buffer$

$box? : BOX\_NAME$

$proj? : PROJ\_ABBR$

$(box?, proj?) \notin known\_boxes$

$\theta cur\_box' = \theta Init\_Box$

$current\_project' = proj?$

$current\_box' = box?$

$current\_part' = stimulus\_response$

$buffer\_access' = modify$

The precondition for this command is that the specified box must not already be known to the system. If the precondition is met, the components of the current box (*cur_box*) are set to the components of *Init_Box* with the box name and project name specified by

the command parameters. The *current_part* is used to describe what part of the box is referenced by the text cursor in the client area of the primary window.

Our first schema does not say what happens if the precondition is not met. For that possibility we use the schema *Box_Exists.*

Box_Exists

$$\Xi Box\_Categories$$
$$box? : BOX\_NAME$$
$$proj? : PROJ\_ABBR$$
$$user\_message! : MESSAGE$$

$$(box?, proj?) \in known\_boxes$$
$$user\_message! = box\_already\_exists$$

We define a special schema *Success* to indicate that a result was *ok.*

Success

$$user\_message! : MESSAGE$$

$$user\_message! = ok$$

If this command is successful the visit to the box is recorded.

Add_Visit

$$\Delta Visit\_Sequence$$
$$box? : BOX\_NAME$$
$$proj? : PROJ\_NAME$$

$$(box?, proj?) \notin \mathrm{rng}\ visits \Rightarrow visits' = visits \frown (box?, proj?)$$

By using the schema calculus, we can now specify a robust version of the *New_Box* command.

$$New\_Box \;\widehat{=}\; (New\_Box\_OK \wedge Success \wedge Add\_Visit) \vee Box\_Exists$$

2.3 *Open_box(box : BOX_NAME, proj : PROJ_ABBR)*

The *Open_Box* command allows the user to recall a previously saved box into the *Buffer* for editing or viewing.

Open_Box_Modify

$$\Xi Projects$$

$\Delta Box\_Categories$

$\Delta Buffer$

$box? : BOX\_NAME$

$proj? : PROJ\_ABBR$

$requested\_access? : ACCESS$

$(box?, proj?) \in available$

$requested\_access? = modify$

$current\_project' = proj?$

$current\_box' = box?$

$current\_part' = stimulus\_response$

$\theta cur\_box' = \theta idBox(box?, proj?)$

$locked\_by' = locked\_by \cup \{(box?, proj?) \mapsto user\}$

$buffer\_access' = modify$

The *Open_Box_Modify* describes a requested open box operation for modify access. If the box is available, it is retrieved into the buffer, the box is locked in the user's name, and the user is granted *modify* access to the buffer.

If the box is locked or reserved, the user may retrieve it for read only access.

Open_Box_Read

$\Xi Projects$

$\Xi Box\_Categories$

$\Delta Buffer$

$box? : BOX\_NAME$

$proj? : PROJ\_ABBR$

$requested\_access? : ACCESS$

$user\_message! : MESSAGE$

$((box?, proj?) \in reserved \cup locked \lor requested\_access? = read)$

$current\_project' = proj?$

$current\_box' = box?$

$current\_part' = stimulus\_response$

$\theta cur\_box' = \theta idBox(box?, proj?)$

$buffer\_access' = read$

$user\_message! = box\_reserved$

The following schemas defines the response for the open box command if the box is

not known to the system.

Box_Not_Known

$$\Xi Box\_Categories$$
$$box? : BOX\_NAME$$
$$proj? : PROJ\_ABBR$$
$$user\_message! : MESSAGE$$

$$(box?, proj?) \notin known\_boxes$$
$$user\_message! = box\_not\_known$$

The complete *Open_Box* specification is given using the schema calculus:

$$Open\_Box \;\widehat{=}\; (Open\_Box\_Modify \wedge Success \wedge Add\_Visit) \vee Open\_Box\_Read \vee Box\_Not\_Known$$

2.4 *Save_Box*

The *Save_Box* command allows the user to save editing changes for the current box that have been made since the last *New_Box* or *Open_Box* command.

Save_Box_OK

$$\Delta Projects$$
$$\Delta Box\_Categories$$
$$\Xi Buffer$$
$$user\_message! : MESSAGE$$

$$buffer\_access = modify$$
$$idBox' = idBox \oplus \{(current\_box, current\_project) \mapsto cur\_box\}$$
$$user\_message! = box\_saved$$

Box_Read_Only

$$\Xi Buffer$$
$$user\_message! : MESSAGE$$

$$buffer\_access = read$$
$$user\_message! = box\_read\_only$$

$$Save\_Box \;\widehat{=}\; Save\_Box\_OK \vee Box\_Read\_Only$$

2.5 *Rename (old_box : BOX_NAME, new_box : BOX_NAME)*

The *Rename* command is used to rename a box including all references to the box.

Rename_Box_OK

ΔBSA

$old\_box? : BOX\_NAME$

$new\_box? : BOX\_NAME$

$user\_message! : MESSAGE$

$old\_box? = current\_box \Rightarrow buffer\_access = modify$

$(old\_box?, current\_project) \in known\_boxes$

$new\_box? \notin known\_boxes^{-1}(\!|\{current\_project?\} \cup reuse\_lib|\!)$

$user\_message! = box\_name\_change$

$idBox' = (old\_box?, current\_project) \lhd idBox \cup$
$\quad \{(new\_box?, current\_project) \mapsto idBox(old\_box?, current\_project)\}$

$known\_boxes' = (known\_boxes \setminus \{(old\_box?, current\_project)\}) \cup$
$\quad \{(new\_box?, current\_project)\}$

$old\_box? = current\_box \Rightarrow current\_box' = new\_box$

Box_is_Reserved

$\Xi Box\_Categories$

$\Xi Buffer$

$box? : BOX\_NAME$

$proj? : PROJ\_ABBR$

$user\_message! : MESSAGE$

$(box?, proj?) \in reserved \cup locked$

$user\_message! = box\_reserved$

To specify the robust version of the *Rename_Box* command we take advantage of the schema renaming facility to enable us to reuse the previously defined *Box_Exists* and *Box_Not_Known* schemas. If S is a schema that contains a variable t then $S[t/u]$ is the schema S with all occurrences of u renamed to t.

$$\begin{aligned} Rename\_Box \;\widehat{=}\; & Rename\_Box\_OK \lor \\ & Box\_is\_Reserved[old\_box?/box?][current\_project/proj?] \lor \\ & Box\_Exists[old\_box?/box?][current\_project/proj?] \lor \\ & Box\_Not\_Known[old\_box?/box?][current\_project/proj?] \end{aligned}$$

2.6 *Delete_Box (box : BOX_NAME, proj : PROJ_NAME)*

A box may only be deleted after all references to the box and all of its descendents have been deleted.

Delete_Box_OK

$$
\begin{array}{l}
\Delta BSA \\
box? : BOX\_NAME \\
proj? : PROJ\_NAME \\
user\_message! : MESSAGE \\
\hline
\forall b : used\_by^{*}(\!|(box?, proj?)|\!) \bullet \\
\quad b \notin \mathrm{rng}((box?, proj?) \lhd all\_uses) \\
box? = current\_box \Rightarrow \\
\quad (current\_box' = \text{“Untitled”} \land \\
\quad current\_part' = null) \\
known\_boxes' = known\_boxes \setminus used\_by^{*}(\!|(box?, proj?)|\!) \\
idBox' = used\_by^{*}(\!|(box?, proj?)|\!) \lhd idBox
\end{array}
$$

The first predicate uses the reflexive-transitive closure function to give the precondition that members of the set of boxes to be deleted can only be used by the specified box or by other descendents of the specified box.

$$Delete\_Box \quad \widehat{=} \quad Delete\_Box\_OK \lor Box\_Not\_Known$$

2.7 *Print_Box (box : BOX_NAME, dev : DEV_TYPE,*
fmt : FORMAT)

The *Print_Box* command formats the contents of a box and prints it on the selected device. Keywords should be bolded. This feature is straight forward and will not be specified formally with the Z notation.

2.8 *Next_Box*

The next box command will select the next box to opened, allow the user to save the current box, and then open the selected box.

Since it is possible that a previously visited box has been deleted either by the current user or concurrently by another user, we must first update our visitation sequence.

Delete_Unknown_Visits

$$
\begin{array}{l}
\Xi Box\_Categories \\
\Delta Visit\_Sequence \\
\hline
visits' = visits \overset{seq}{\rhd} known\_boxes
\end{array}
$$

The following schema determines the next box to select:

Select_Next_Box

$\Xi Visit\_Sequence$

$\Xi Buffer$

$box! : BOX\_NAME$

$proj! : PROJ\_ABBR$

$((current\_box, current\_project) \in visits(\!| 1 \,..\, (\#visits - 1) |\!) \land$

$\quad (box!, proj!) = visits(visits^{-1}(current\_box, current\_project) + 1)) \quad \lor$

$((current\_box, current\_project) = visits(\#visits) \land$

$\quad (box!, proj!) = visits(1))$

If a next box is successfully selected the user will be prompted to save the current box.

Save_Current

$\Xi Buffer$

$save\_request? : BUTTON$

$buffer\_access = modify$

$save\_request? = yes$

If *Op1* and *Op2* are schemas then *Op1* ; *Op2* describes their *sequential composition.* This is a composite operation in which first *Op1* then *Op2* occurs. The full *Next_Box* command is a composition of several operations. First the deleted boxes are removed from the visit list, then the next box is selected. The user is then given an opportunity to save the current box before the selected box is opened.

$$Next\_Box \;\widehat{=}\; Delete\_Unknown\_Visits \,;\, Select\_Next\_Box \,;\, (Save\_Current \land Save\_Box) \,;\, Open\_Box$$

2.9 *Prev_Box*

Select_Prev_Box

$\Xi Visit\_Sequence$

$\Xi Buffer$

$box! : BOX\_NAME$

$proj! : PROJ\_ABBR$

$((current\_box?, current\_project?) \in visits(\!| 2 \,..\, \#visits |\!) \land$

$\quad (box!, proj!) = visits(visits^{-1}(current\_box, current\_project) - 1)) \quad \land$

$((current\_box?, current\_project?) = visits(1) \land$

$$(box!, proj!) = visits(\#visits))$$

$$Prev\_Box \;\widehat{=}\; Delete\_Unknown\_Visits \,;\, Select\_Prev\_Box \,;\, (Save\_Current \wedge Save\_Box) \,;\, Open\_Box$$

2.10 *Exit_Session*

When the user selects the *Exit_Session* command the system will prompt the user whether to save the current box, then the *exit_form* will be displayed before the system is exited. The system must remember the current project for the next session by this user.

Exit_Session_OK

$$\Xi Buffer$$
$$\Delta User\_Info$$
$$disp\_form! : DISPLAYS$$

$$user\_project' = user\_project \oplus \{usr? \mapsto current\_project\}$$
$$disp\_form! = exit\_form$$

The full command is given using schema composition.

$$Exit\_Session \;\widehat{=}\; (Save\_Current \wedge Save\_Box) \,;\, Exit\_Session\_OK$$

3. Edit Menu Commands

The edit menu commands allow the user to transfer information between the current box being edited and the system *clipboard*. The box being edited and the clipboard can be ultimately be viewed as a sequence of characters.* The box text being manipulated can be divided into three subsequences: *left*, *selection*, and *right*. The *selection* represents either a selected area of the text or, if no selection is active, the last character that was inserted into the document. The clipboard contains data that was cut or copied from the text. The previous contents of the clipboard must also be represented to accomodate the *Undo Cut* command.

Box_Text

$$text, left, right, selection : \text{seq } CH$$
$$clipboard : \text{seq } CH$$
$$prev\_clipboard : \text{seq } CH$$
$$last\_edit\_cmd : ED\_CMD$$

*The idea of modeling editor functions using sequences of characters is based upon work by Sufrin [26,27].

$$
\begin{array}{|l}
text\_type : ED\_TYPE \\
clipboard\_type : ED\_TYPE \\
prev\_clipboard\_type : ED\_TYPE \\
\hline
text = left \frown selection \frown right \\
\hline
\end{array}
$$

Information in the clipboard must be compatible with the kind of text being edited before the clipboard information can be copied into the text. For instance, a clipboard of type *proc* would not be compatible with text of type *data*. To specify this compatibility we define the global constant relation *is_compatible*.

$$
\begin{array}{|l}
is\_compatible : ED\_TYPE \leftrightarrow ED\_TYPE \\
\hline
is\_compatible = \{text \mapsto text, data \mapsto text, proc \mapsto text, \\
\qquad\qquad data \mapsto data, proc \mapsto proc\}
\end{array}
$$

Initially, *text*, *prev_clipboard*, and *clipboard* are empty.

Init_Box_Text

$$
\begin{array}{|l}
Box\_Text \\
\hline
text = clipboard = prev\_clipboard = \langle\rangle \\
last\_edit\_cmd = nil \\
clipboard\_type = prev\_clipboard\_type = nil \\
\hline
\end{array}
$$

3.1 *Edit_Menu*

Edit_Menu

$$
\begin{array}{|l}
disp\_form! : DISPLAY \\
\hline
disp\_form! = edit\_menu \\
\hline
\end{array}
$$

3.2 *Undo*

The *Undo* command reverses the most recently executed user action of *Cut*, *Paste*, *Insert Construct*, or *Modify Construct*.

$\langle Undo\_Cut \rangle$ re-inserts text that was previously cut, selects the text, and the contents of the clipboard are restored to what it contained prior to the cut.

Undo_Cut

$$
\begin{array}{|l}
\Delta Box\_Text \\
\hline
last\_edit\_cmd = cut \\
selection = \langle\rangle
\end{array}
$$

$$
\begin{array}{l}
clipboard\_type\ is\_compatible\ text\_type \\
left' = left \\
right' = right \\
selection' = clipboard \\
clipboard' = prev\_clipboard \\
clipboard\_type = prev\_clipboard\_type \\
last\_edit\_cmd = nil
\end{array}
$$

The $\langle Undo\_Copy \rangle$ command restores the *selection* and the *clipboard* but does not change the *text*.

Undo_Copy

$$
\begin{array}{|l}
\Delta Box\_Text \\
\hline
last\_edit\_cmd = copy \\
selection = \langle\rangle \\
left \hookrightarrow clipboard \\
left' = truncate(left, clipboard) \\
selection' = clipboard \\
text' = text \\
clipboard' = prev\_clipboard \\
clipboard\_type' = prev\_clipboard\_type \\
last\_edit\_cmd' = nil
\end{array}
$$

The $\langle Undo\_Paste \rangle$ removes the text that was previously pasted and restores the selection.

Undo_Paste

$$
\begin{array}{|l}
\Delta Box\_Text \\
\hline
last\_edit\_cmd = paste \\
selection \neq \langle\rangle \\
selection \hookrightarrow clipboard \\
selection' = truncate(selection, clipboard) \\
left' = left \\
right' = right \\
clipboard' = clipboard \\
last\_edit\_cmd' = nil
\end{array}
$$

3.3 *Cut*

The $\langle Cut \rangle$ command copies the selected sequence to the clipboard and then removes it from the object being edited.

Cut

ΔBox_Text

$selection \neq \langle\rangle$

$selection' = \langle\rangle$

$left' = left$

$right' = right$

$clipboard' = selection$

$prev\_clipboard' = clipboard$

$prev\_clipboard\_type' = clipboard\_type$

$clipboard\_type' = text\_type$

$last\_edit\_cmd' = cut$

3.4 *Copy*

The $\langle Copy \rangle$ command copies the selection to the clipboard without removing the selected text from the object being edited.

Copy

ΔBox_Text

$selection \neq \langle\rangle$

$text' = text$

$left' = left \frown selection$

$selection' = \langle\rangle$

$clipboard' = selection$

$prev\_clipboard' = clipboard$

$prev\_clipboard\_type' = clipboard\_type$

$clipboard\_type' = text\_type$

$last\_edit\_cmd' = copy$

3.5 *Paste*

The paste command copies the contents of the clipboard into the object being edited at the current cursor position.

Paste

ΔBox_Text

$clipboard\_type\ is\_compatible\ text\_type$

$left' = left \frown selection$

$right' = right$

$selection' = clipboard$

$clipboard' = clipboard$

$prev\_clipboard' = selection$

$prev\_clipboard\_type' = text\_type$

$last\_edit\_cmd' = paste$

4. Project Operations

The project operations allow the user to select actions to create, modify, delete projects. A project usage hierarchy report can be produced. The user can also reserve, replace and unreserve boxes in the project library.

4.1 *Project_Menu*

Project_Menu

$disp\_form! : DISPLAY$

$disp\_form! = project\_menu$

4.2 *New_Project (proj : PROJ_ABBR, reuse : REUSE_IND)*

The *New_Project* command creates a new project and makes it the current project for this user.

New_Project_OK

$\Delta Projects$

$\Delta Buffer$

$proj? : PROJ\_ABBR$

$reuse? : REUSE\_IND$

$user\_message! : MESSAGE$

$proj? \notin known\_projects$

$(reuse? = yes \Rightarrow$

$\quad reuse\_lib' = reuse\_lib \cup \{proj?\} \vee$

$reuse? = no \Rightarrow$

$\quad proj\_lib' = proj\_lib \cup \{proj?\})$

$current\_project' = proj?$

$current\_box' =$ "Untitled"

$current\_part = null$

$\theta cur\_box = \theta Init\_box$

$buffer\_access' = read$

$user\_message! = new\_project\_confirm$

Project_Exists

$\Xi Projects$

$proj? : PROJ\_ABBR$

$reuse? : REUSE\_IND$

$user\_message! : MESSAGE$

$(proj?, reuse?) \in known\_projects$

$user\_message! = project\_already\_known$

$$New\_Project \;\widehat{=}\; ((Save\_Current \wedge Save\_Box) \,;\, New\_Project\_OK) \vee Project\_Exists$$

4.3 *Open_Project (proj : PROJ_ABBR)*

The open project command allows the user to open an existing project.

Open_Project_OK

$\Xi Projects$

$\Delta Buffer$

$proj? : PROJ\_ABBR$

$user\_message! : MESSAGE$

$proj? \in known\_projects$

$proj? \neq current\_project$

$current\_project' = proj?$

$current\_box' =$ "Untitled"

$current\_part = null$

$\theta cur\_box = \theta Init\_box$

$buffer\_access' = read$

$user\_message! = open\_project\_confirm$

Project_Already_Open

$\Xi Buffer$

$proj? : PROJ\_ABBR$

$user\_message! : MESSAGE$

$proj? = current\_project$

$user\_message! = project\_already\_open$

Project_Not_Known

$\Xi Projects$

$proj? : PROJ\_ABBR$

$user\_message! : MESSAGE$

$proj? = current\_project$

$user\_message! = project\_not\_known$

$$Open\_Project \;\widehat{=}\; ((Save\_Current \wedge Save\_Box) \mathbin{;} Open\_Project\_OK) \vee Project\_Already\_Open \vee Project\_Not\_Known$$

4.4 *Delete_Project (proj : PROJ_ABBR)*

The user will be allowed to delete a project as long as none of the boxes in the project's library are locked, reserved, or are used by other projects.

Delete_Project_OK

ΔBSA

$\Delta Buffer$

$proj? : PROJ\_ABBR$

$user\_message! : MESSAGE$

$proj? \in known\_projects$

$proj? \notin \mathrm{rng}\,(locked \cup reserved)$

$proj? \in reuse\_lib \Rightarrow$

$\quad \forall b : box\_reuse \mid second\ b = proj? \bullet$

$\quad\quad used\_by^{+}(\!|b|\!) \rhd \{proj?\} = \emptyset$

$known\_projects' = known\_projects \setminus \{proj?\}$

$known\_boxes' = known\_boxes \rhd \{proj?\}$

$user\_message! = delete\_confirmation$

The third predicate says that if the project is a reuse library, then no box in the library may be used by other projects. If the project being deleted is the current project, then it must be cleared from the buffer.

Clear_Current_Project

$\Delta Buffer$

$proj? : PROJ\_ABBR$

$proj? = current\_project$

$current\_project' =$ "Untitled"

$current\_box' =$ "Untitled"

$current\_part' = null$

$buffer\_access' = read$

$\theta cur\_box' = \theta Init\_Box$

The user is not allowed to delete projects if any of the project's boxes are locked or reserved.

Proj_Boxes_Inuse

$\Xi Box\_Categories$

$proj? : PROJ\_ABBR$

$user\_message! : MESSAGE$

$proj? \in \text{rng}\,(locked \cup reserved)$

$user\_message! = proj\_boxes\_in\_use$

$$Delete\_Project \;\widehat{=}\; (Delete\_Project\_OK \wedge Clear\_Current\_Project) \vee Project\_Not\_Known \vee Proj\_Boxes\_Inuse$$

4.5 *Print_Hierarchy (device : DEV_TYPE, fmt : FORMAT)*

The *Print_Hierarchy* command prints an indentation chart for a project. The first line prints the top-level box of the project. Then the hierarchy is printed for each of the used boxes of the top-level box. The child boxes of each parent are those specified by the *boxes_used* injective sequence. The number of tabs that are used to indent each box usage line is equal to the box's depth in the usage hierarchy tree. The sequence of lines printed is determined by a depth first recursive printing order. To print a usage hierarchy for a box, first print the line for a box and increment the tab level. Then print a usage hierarchy for each box in the sequence of boxes used by that box. Finally, decrement the tab level.

For convenience, instead of using standard Z, we specify this function using a recursive BDL algorithm which references the Z defined state data.

print_hierarchy($box : BOX\_ID, d : \mathbb{N}$)

$\Xi Projects$

$\Xi Usage\_Hierarchy$

$hier\_line! : \text{seq } CH$

$hier\_line! = tabs_d \frown \langle first\ box \rangle$

$i := 1$

do while $i < \#(idBox(box).boxes\_used)$

  let $b = idBox(box).boxes\_used(i)$

  print_hierarchy$(b, d + 1)$

  $i := i + 1$

od

where $tabs_k = \lambda n : \mathbb{N} \mid n \in 1 \,..\, k \bullet tab$

4.6 *Reserve_Box (box : BOX_NAME, proj : PROJ_ABBR)*
This command reserves a specified box in the project library under the current username.

Reserve_Box_OK

$\Delta Box\_Categories$

$\Xi Buffer$

$box? : BOX\_NAME$

$proj? : PROJ\_ABBR$

$user\_message! : MESSAGE$

$(box?, proj?) \in known\_boxes$

$(box?, proj?) \in available$

$reserved\_by' = reserved\_by \oplus \{(box?, proj?) \mapsto user\}$

$user\_message! = box\_reserved$

When a box is reserved, the contents of the box are copied to a file in the user's current directory. An output file consists of a sequence of bytes:

$$FILE \mathrel{\widehat=} \text{seq } BYTE$$

The file storage system (SS) on a computer can be viewed as a partial function from file identifiers (FID) to FILE. For our purposes, the name of the file identifier assigned to an exported file will be the name of the box being exported, with a "`.CDL`" extension.

SS

$fstore : FID \pfun FILE$

We leave the format of this file to be determined by the developers. However, it is an obvious requirement that a reserve box command immediately followed by a replace box command should cause no change to the system state.

Export_Box

$\Xi Projects$

$\Xi Buffer$

ΔSS

$box? : BOX\_NAME$

$proj? : PROJ\_ABBR$

---

let $fname = \langle box? \rangle \frown$ ".CDL"

$fstore' = fstore \oplus \{fname \mapsto cvt2file(idBox(box?, proj?))\}$

where *cvt2file is [to be determined]*

We must specify what happens if the box is already reserved or if it is locked by another user.

Box_Already_Reserved

$\Xi Box\_Categories$

$box? : BOX\_NAME$

$proj? : PROJ\_ABBR$

$user\_message! : MESSAGE$

$(box?, proj?) \in reserved$

$user\_message! = box\_already\_reserved$

Box_Inuse

$\Xi Box\_Categories$

$\Xi Buffer$

$box? : BOX\_NAME$

$proj? : PROJ\_ABBR$

$user\_message! : MESSAGE$

$(box?, proj?) \in locked\_by \rhd \{user?\}$

$user\_message! = box\_locked$

We can now use the schema calculus to specify the robust *Reserve_Box* command:

$$Reserve\_Box \;\widehat{=}\; (Reserve\_Box\_OK \wedge Export\_Box) \vee Box\_Already\_Reserved \vee Box\_Inuse \vee Box\_Not\_Known$$

4.7 *Unreserve_Box (box : BOX_NAME, proj : PROJ_ABBR)*
This command allows a user to remove a box from reserve status without changing it's contents.

Unreserve_Box_OK

$\Delta Box\_Categories$
$\Xi Buffer$
$box? : BOX\_NAME$
$proj? : PROJ\_ABBR$
$user\_message! : MESSAGE$

$(box?, proj?) \in known\_boxes$
$user = reserved\_by(box?, proj?)$
$reserved\_by' = (box?, proj?) \lhd reserved\_by$
$user\_message! = box\_unreserved$

It is possible the box is reserved by a different user or is not reserved at all.

Reserved_By_Another

$\Xi Box\_Categories$
$\Xi Buffer$
$box? : BOX\_NAME$
$proj? : PROJ\_ABBR$
$user\_message! : MESSAGE$

$(box?, proj?) \in known\_boxes$
$(box?, proj?) \in \mathrm{dom}\,(reserved\_by \rhd user)$
$user\_message! = box\_reserved$

Box_Not_Reserved

$\Xi Box\_Categories$
$box? : BOX\_NAME$

$proj? : PROJ\_ABBR$
$user\_message! : MESSAGE$

$(box?, proj?) \in known\_boxes$
$(box?, proj?) \notin reserved$
$user\_message! = box\_not\_reserved$

$$Unreserve\_Box \;\widehat{=}\; Unreserve\_Box\_OK \lor Reserved\_by\_Another \lor Box\_Not\_Reserved \lor Box\_Not\_Known$$

4.8 *Replace_Box (box : BOX_NAME, proj : PROJ_ABBR)*
The replace command returns a previously reserved box into the library.

Replace_Box_OK

$\Xi Buffer$
ΞSS
$box? : BOX\_NAME$
$proj? : PROJ\_NAME$
$user\_message! : MESSAGE$

$(box?, proj?) \in known\_boxes$
$reserved\_by(box?, proj?) = user$
$reserved\_by' = (box?, proj?) \lhd reserved\_by$
$user\_message! = box\_replaced$

Import_Box

$\Delta Projects$
$\Xi Buffer$
ΞSS
$box? : BOX\_NAME$
$proj? : PROJ\_ABBR$

---

let $fname = \langle box? \rangle \frown$ ".CDL"
$fname \in \mathrm{dom}\, fstore$
$is\_valid\_box((box?, proj?), fstore(fname)) = true$
$idBox' = idBox \oplus \{(box?, proj?) \mapsto cvt2box(idBox(box?, proj?))\}$

where *cvt2box is [to be determined] and*
is_valid_box is [to be determined]

The next schema accounts for the possibility that the box fails the validation tests:

Invalid_Box

$\Xi Box\_Categories$
ΞSS
$box? : BOX\_NAME$
$proj? : PROJ\_ABBR$
$user\_messages! = \text{seq}\ MESSAGE$

$(box?, proj?) \in known\_boxes$
$reserved\_by(box?, proj?) = user$
let $fname = \langle box? \rangle \frown$ ".CDL"
$fname \in \text{dom}\ fstore$
$is\_valid\_box((box?, proj?), fstore(fname)) = false$
$user\_messages! = \langle error\ messages\ [tbd] \rangle$

Another problem that could occur is that the file does not exist.

File_Not_Found

$\Xi Box\_Categories$
ΞSS
$box? : BOX\_NAME$
$proj? : PROJ\_ABBR$
$user\_message! = MESSAGE$

$(box?, proj?) \in known\_boxes$
$reserved\_by(box?, proj?) = user$
let $fname = \langle box? \rangle \frown$ ".CDL"
$fname \notin \text{dom}\ fstore$
$user\_message! = file\_not\_found$

$$Replace\_Box \ \widehat{=}\ (Replace\_Box\_OK \wedge Import\_Box) \vee Invalid\_Box \vee File\_Not\_Found \vee Reserved\_by\_Another \vee Box\_Not\_Reserved \vee$$

Box_Not_Known

5. Part Operations

This group of operations allows the user to navigate among box parts.

5.1 *Part*

Part_Menu

$$disp\_form! : DISPLAY$$

$$disp\_form! = part\_menu$$

5.2 *Next_Part*

For this operation we utilized the previously defined constant function *next_part*.

Next_Part_OK

$$\Delta Buffer$$

$$current\_part \neq null$$
$$current\_part' = next\_part(current\_part)$$

Box_Not_Open

$$\Xi Buffer$$
$$user\_message! : MESSAGE$$

$$current\_part = null$$
$$user\_message! = box\_not\_open$$

$$Next\_Part \quad \widehat{=} \quad Next\_Part\_OK \vee Box\_Not\_Open$$

5.3 *Prev_Part*

For this operation we utilized the previously defined constant function *prev_part*.

Prev_Part_OK

$$\Delta Buffer$$

$$current\_part \neq null$$
$$current\_part' = prev\_part(current\_part)$$

$$Prev\_Part \quad \widehat{=} \quad Prev\_Part\_OK \vee Box\_Not\_Open$$

5.4 *Stimulus_response*

Stimulus_response_OK

$\Delta Buffer$

$current\_part \neq null$

$current\_part' = stimulus\_response$

$$Stimulus\_response \;\widehat{=}\; Stimulus\_response\_OK \vee Box\_Not\_Open$$

5.5 *Transition*

Transition_OK

$\Delta Buffer$

$current\_part \neq null$

$current\_part' = transition$

$$Transition \;\widehat{=}\; Transition\_OK \vee Box\_Not\_Open$$

5.6 *State*

State_OK

$\Delta Buffer$

$current\_part \neq null$

$current\_part' = state$

$$State \;\widehat{=}\; State\_OK \vee Box\_Not\_Open$$

5.7 *Machine*

Machine_OK

$\Delta Buffer$

$current\_part \neq null$

$current\_part' = machine$

$$Machine \;\widehat{=}\; Machine\_OK \vee Box\_Not\_Open$$

5.8 *Data*

Data_OK

$\Delta Buffer$

$current\_part \neq null$

$current\_part' = data$

$$Data \;\widehat{=}\; Data\_OK \vee Box\_Not\_Open$$

5.9 *Procedure*

Procedure_OK

$\Delta Buffer$

$current\_part \neq null$

$current\_part' = procedure$

$$Procedure \;\widehat{=}\; Procedure\_OK \vee Box\_Not\_Open$$

6. Structure Menu Operations

The structure menu provides user commands to insert and modify syntax structures within a clear box procedure.

6.1 *Structure*

The structure menu is only available when the user is working with a clear box procedure part and has modify access to the box.

Structure_Menu_OK

$\Xi Buffer$

$disp! = DISPLAY$

$current\_part = procedure$

$buffer\_access = modify$

$disp! = structure\_menu$

Clear_Box_Only

$\Xi Buffer$

$user\_message! : MESSAGE$

$current\_part \neq procedure$

$user\_message! = only\_in\_proc$

$Structure\_Menu \mathrel{\widehat=} Structure\_Menu\_OK \lor Box\_Read\_Only \lor Clear\_Box\_Only$

6.2 *Insert*

The insert submenu allows the user to list an item menu containing CDL syntax structures that may be inserted into the clear box procedure.

Insert_Menu_OK

$\Xi Buffer$

$disp! : DISPLAY$

$current\_part = procedure$

$buffer\_access = modify$

$disp! = insert\_menu$

$Insert\_Menu \mathrel{\widehat=} Insert\_Menu\_OK \lor Box\_Read\_Only \lor Clear\_Box\_Only$

6.3 *use_BB*

The *use_BB* selection from the *Insert_Menu* will display a template of the box. If the box doesn't exist, the user must first create the interface to the box.

Create_Box_Def

ΔBSA

$\Xi Buffer$

$box? : BOX\_NAME$

$args? : \mathrm{iseq}\,(IDENT \times (MODE \times TYPE))$

$user\_message! : MESSAGE$

$box? \notin known\_boxes^{-1}(\{current\_project\})$

$box? \notin \mathrm{dom}\ box\_reuse$

$idBox' = idBox \oplus \{(box?, current\_project) \mapsto new\_box\}$

$user\_message! = new\_box\_confirm$

where $new\_box : Box \bullet$

$box\_name = box?$

$proj\_abbr = current\_project$

$stim\_resp = strip(args?)$

$$
\begin{aligned}
&interface = strip(args?)\\
&transtion = \langle\rangle\\
&machine = \langle\rangle\\
&procedure = nilP\\
&box\_data = \mathrm{rng}\, strip(args?)\\
&data\_access\_mode = \{i : \mathrm{rng}\, args? \bullet first(i) \mapsto first(second(i))\}\\
&data\_type = \{i : \mathrm{rng}\, args? \bullet first(i) \mapsto second(second(i))\}\\
&data\_comments = box\_calls = \emptyset
\end{aligned}
$$

We rely upon invariant conditions to insert this box definition into the set of known boxes. Note that in the **where** clause we allow ourselves to omit the schema selector (e.g., *new_box.box_name*) since in this context it should be clear that we are referring to the components of *new_box*.

Once the box definition has been created, the steps for inserting a *use_BB* structure are the same as for an existing box. The next schema defines the constraints upon the list of variables that a user can supply to a *use_BB* template.

Use_Existing_BB

$$
\begin{aligned}
&\Delta Buffer\\
&\Xi BSA\\
&box? : BOX\_NAME\\
&proj' : PROJ\_ABBR\\
&box\_template' : \mathrm{seq}\,(TYPE \times MODE)\\
&\hline
&box? \in known\_boxes^{-1}(\!|\{current\_project\}|\!) \Rightarrow\\
&\qquad proj' = current\_project\\
&box? \in \mathrm{dom}\, box\_reuse \Rightarrow\\
&\qquad proj' = box\_reuse(box?)\\
&\textbf{let}\ ub = idBox(box?, proj!)\\
&\textbf{let}\ arg = ub.interface\\
&box\_template' = \lambda n : N \mid n \in 1\,..\,\#arg \bullet\\
&\quad (ub.data\_type(arg(n)), ub.data\_access\_mode(arg(n)))
\end{aligned}
$$

The preconditions in the next schema define the validation of arguments for a *use_BB* template given an argument list to validate and a box template to validate against. The length of the argument list and the box template must be equal. Every identifier in the argument list must be defined in the box data for the current box. The data type of every argument must match the data types of the template. Also, if an identifier has a local data access mode of read-only (*in*) then the mode of the interface must also be read-only.

The postcondition partially describe how the current box (*cur_box*) is updated with the new template.

Update_Use_BB

$$
\begin{array}{l}
\Delta Buffer \\
box? : BOX\_NAME \\
arg\_list? : \mathrm{seq}\, IDENT \\
box\_template : \mathrm{seq}\,(TYPE \times MODE) \\
\hline
\#arg\_list? = \#box\_template \\
\mathrm{rng}\, arg\_list? \subseteq cur\_box.box\_data \\
\forall n : \mathbb{N} \mid n \in 1 \,..\, \#arg\_list? \bullet \\
\quad (cur\_box.data\_type(arg\_list?(n)) = first(box\_template(n)) \land \\
\quad cur\_box.data\_access\_mode = in \Rightarrow \\
\qquad second(arg\_list?(n)) = in) \\
cur\_box.box\_calls' = cur\_box.box\_calls \frown \langle (box?, proj?) \mapsto arg\_list? \rangle
\end{array}
$$

$$
\begin{array}{lcl}
Use\_BB & \widehat{=} & ((Create\_Box\_Def \,;\, Use\_Existing\_BB) \lor Use\_Existing\_BB) \,; \\
 & & Update\_Use\_BB
\end{array}
$$

6.4 *Modify*

The modify submenu allows the user to list an item menu containing CDL syntax structures that may be used to modify syntax structures in the clear box procedure.

Modify_Menu_OK

$$
\begin{array}{l}
\Xi Buffer \\
disp! : DISPLAY \\
\hline
current\_part = procedure \\
buffer\_access = modify \\
disp! = modify\_menu
\end{array}
$$

$$Modify\_Menu \mathrel{\widehat{=}} Modify\_Menu\_OK \lor Box\_Read\_Only \lor Clear\_Box\_Only$$

7. Options Menu

7.1 *Options*

The options menu allows the user to customize the application in various ways.

Options_Menu_OK

$\Xi Buffer$

$disp! : DISPLAY$

$current\_part = procedure$

$buffer\_access = modify$

$disp! = options\_menu$

$$Options\_Menu \mathrel{\widehat=} Options\_Menu\_OK \lor Box\_Read\_Only \lor Clear\_Box\_Only$$

The commands in the option menu will not be specified formally since they deal mainly with the user interface and also because they are not considered an essential part of the system.

7.2 *New_Structure_Pallette*
This option lets the user choose whether the insertion item menu will remain on the display at all times while editing the clear box procedure.

7.3 *Modify_Structures_Pallette*
This option allows the user to keep a display on the screen listing all possible commands for modifying syntax structures.

7.4 *Visitation_Rules*
Invokes a dialogue box to allow the user to specify options for box visitation order.

8. Help Menu

For the help menu options we assume that help information may be recorded for some or all of the fields and windows. The following state schema will serve as a repository of this help information.

Help_Information

$known\_fields : \mathbb{P}\ IDENT$

$known\_windows : \mathbb{P}\ IDENT$

$known\_keys : \mathbb{P}\ IDENT$

$field\_help : IDENT \pfun TEXT$

$window\_help : IDENT \pfun TEXT$

$keys\_help : IDENT \pfun TEXT$

$help\_help : TEXT$

$\mathrm{dom}\ field\_help \subseteq known\_fields$

$\mathrm{dom}\ window\_help \subseteq known\_windows$

$$\text{dom } keys\_help = known\_keys$$
$$help\_help \neq \langle\rangle$$

We assume the maintenance of this help information is performed by the developers using text editors or other development tools. Note that in our invariant conditions we allow help information to be missing for some of the fields and windows. However, help should be provided on how to use the help facility and also on all key assignments used within the application.

8.1 *Context_Help*

This command provides specific information about the item on which the selection cursor is positioned.

Context_Help_OK

$$\Xi Help\_Information$$
$$current\_field? : IDENT$$
$$help\_display! : TEXT$$

$$current\_field? \in \text{dom } field\_help$$
$$help\_display! = field\_help(current\_field?)$$

No_Context_Help

$$\Xi Help\_Information$$
$$current\_field? : IDENT$$
$$user\_message! : MESSAGE$$

$$current\_field? \notin \text{dom } field\_help$$
$$user\_message! = no\_field\_help\_avail$$

$$Context\_Help \;\widehat{=}\; Context\_Help\_Ok \lor No\_Context\_Help$$

8.2 *Help*

This command is used to simply display the help menu.

Help

$$disp! : DISPLAY$$

$$disp! = help\_menu$$

8.3 *Extended_Help*

This command requests information about the tasks that can be performed in the current application window.

Extended_Help_OK

$$\Xi Help\_Information$$
$$current\_window? : IDENT$$
$$help\_display! : TEXT$$

$$current\_window? \in \text{dom } window\_help$$
$$help\_display! = window\_help(current\_window?)$$

No_Extended_Help

$$\Xi Help\_Information$$
$$current\_window? : IDENT$$
$$user\_message! : MESSAGE$$

$$current\_window? \notin \text{dom } window\_help$$
$$user\_message! = no\_window\_help\_avail$$

$$Extended\_Help \;\widehat{=}\; Extended\_Help\_Ok \vee No\_Extended\_Help$$

8.4 *Help_Help*

This command displays information about how to use the help facility.

Help_Help

$$\Xi Help\_Information$$
$$help\_display! : TEXT$$

$$help\_display! = help\_help$$

8.5 *Keys_Help*

Display information describing the key assignments of the BSA application.

Keys_Help

$$\Xi Help\_Information$$
$$key? : IDENT$$
$$help\_display! : TEXT$$

$$help\_display! = keys\_help(key?)$$

9. Data Item Form Operations

Definitions: We will model a *Data_Item_Form* as a pair of finite sequences of records where each record consists of a pair of finite sequences of fields and each field consists of a pair of finite sequences of characters. Pairs of sequences are used to model the user's cursor position within the form. In each case, the left element of the pair represents objects (records, fields or characters) prior to the user's current cursor position and the right element of the pair represents objects on or after the current cursor position. The cursor can be considered to be pointing to the head of the right element in the pair.

Text_Field

$$
\begin{array}{l}
fld\_len : \mathbb{N} \\
text : \text{seq } CH \\
left : \text{seq } CH \\
right : \text{seq } CH \\
\hline
text = left \frown right \\
\#text \leq fld\_len
\end{array}
$$

The length of each field must be less than or equal to a specified maximum length.

Record

$$
\begin{array}{l}
record : \text{seq}(Text\_Field) \\
left\_fields : \text{seq}(Text\_Field) \\
right\_fields : \text{seq}(Text\_Field) \\
\hline
record = left\_fields \frown right\_fields \\
\forall t : \text{rng } left\_fields \bullet t.left = \langle\rangle \\
\forall t : \text{rng } tail(right\_fields) \bullet t.left = \langle\rangle
\end{array}
$$

The final two invariant conditions specify that the starting character position of every field other than the current field is the far left character in that field.

Data_Item_Form

$$
\begin{array}{l}
num\_fields : \mathbb{N} \\
table : \text{seq } Record \\
left\_rows : \text{seq } Table\_Row \\
right\_rows : \text{seq } Table\_Row \\
\hline
table = left\_rows \frown right\_rows \\
\forall r : \text{rng } table \bullet \#r.record = num\_fields \\
\forall r : \text{rng } left\_rows \bullet r.left\_fields = \langle\rangle
\end{array}
$$

$$\forall r : \mathrm{rng}\ tail(right\_rows) \bullet r.left\_fields = \langle\rangle$$

The second invariant condition requires that every record in the form must contain the same number of fields. The final two invariant conditions specify that the starting field of each record other than the current record is the far left field.

9.1 *Insert_ch*

Insert_ch

$$\Delta Text\_Field$$
$$\Xi Buffer$$
$$ch? : CH$$

$$buffer\_access = modify$$
$$entry\_mode = insert$$
$$left' = left \frown \langle ch? \rangle$$
$$right' = right$$

If the additional character would cause the field to overflow, an error message is issued.

Field_Is_Full

$$\Xi Text\_Field$$
$$\Xi Buffer$$
$$user\_message! : MESSAGE$$

$$buffer\_access = modify$$
$$\#text = fld\_len$$
$$entry\_mode = insert \lor right = \langle\rangle$$
$$user\_message! = field\_is\_full$$

9.2 *Overstrike_ch*

Overstrike_ch

$$\Delta Text\_Field$$
$$\Xi Buffer$$
$$ch? : CH$$

$$buffer\_access = modify$$
$$entry\_mode = overstike$$
$$left' = left \frown \langle ch? \rangle$$

$$right' = tail(right)$$

The operation of typing a character inside a data item form is partially described by the schema *Enter_ch*.

$$Enter\_ch \;\widehat{=}\; ((Insert\_ch \lor Overstrike\_ch) \land Success) \lor Field\_Is\_Full \lor Box\_Read\_Only$$

9.3 *Ins_Ovr_Toggle*

Ins_Ovr_Toggle

$\Delta Buffer$

$entry\_mode = insert \Rightarrow entry\_mode' = overstrike$

$entry\_mode = overstrike \Rightarrow entry\_mode' = insert$

9.4 *Move*

This family of operations allows the user to move the cursor in either direction from one character, field, or record to the next one.

Move_right_ch

$\Delta Text\_Field$

$text' = text$

$right' = tail(right)$

Move_left_ch

$\Delta Text\_Field$

$text' = text$

$left' = front(left)$

Move_right_field

$\Delta Record$

$record' = record$

$right\_fields' = tail(right\_fields)$

Move_left_field

$\Delta Record$

$record' = record$

$left\_fields' = front(left\_fields)$

Move_right_record

$\Delta Data\_Item\_Form$

$table' = table$

$right\_rows' = tail(right\_rows)$

Move_left_record

$\Delta Data\_Item\_Form$

$table' = table$

$left\_rows' = front(left\_rows)$

9.5 *Del_left_ch*

Del_left_ch

$\Delta Text\_Field$

$\Xi Buffer$

$buffer\_access = modify$

$left' = front(left)$

$right' = right$

9.6 *Del_right_ch*

Del_right_ch

$\Delta Text\_Field$

$\Xi Buffer$

$buffer\_access = modify$

$right' = tail(right)$

$left' = left$

9.7 *Blank_field*

Blank_field

$\Delta Text\_Field$

$\Xi Buffer$

$buffer\_access = modify$

$text' = \langle\rangle$

9.8 *Del_rec*

Del_rec

$\Delta Data\_Item\_Form$

$\Xi Buffer$

$buffer\_access = modify$

$left\_rows' = left\_rows$

$right\_rows' = tail(right\_rows)$

9.9 *Open_left_rec*

To open a row requires that we initialize a new record to default values. We use the schema *New_Rec* to specify the initial state of a newly inserted record.

New_Rec

Record

$left\_fields = \langle\rangle$

$\forall f : \text{rng } record \bullet f.text = \langle\rangle$

All fields in a new record are initially set to empty sequences. By setting *left_fields* to the empty sequence we are specifying that the cursor will be positioned to the left most field.

Open_left_rec

$\Delta Data\_Item\_Form$

$\Xi Buffer$

New_Rec

$buffer\_access = modify$

$left\_rows' = left\_rows$

$right\_rows' = \langle New\_Rec.record\rangle \frown right\_rows$

9.10 *Pick_item*

This user operation is designed for entering a value into the data type field of a data item form from a pick list. To model this command we must first specify a schema for the pick list.

Pick_List

$$item\_list : \text{iseq } TEXT$$
$$num\_items : \mathbb{N}$$
$$\#item\_list = num\_items$$

Pick_item

$$\Delta Text\_Field$$
$$\Xi Buffer$$
$$\Xi Pick\_List$$
$$n? : \mathbb{N}$$
$$buffer\_access = modify$$
$$text' = item\_list(n?)$$
$$left' = \langle\rangle$$

10. Text Editor Operations

Text editor commands have some of the same operations as the data item form. For these operations we shall use the *Box_Text* schema defined for the edit menu commands.

10.1 *Entering characters*

The operations of inserting or overstriking characters into a text field are given as follows:

$$Enter\_ch\_ed \;\widehat{=}\; Enter\_ch[Box\_Text/Text\_Field]$$

10.2 *Move*

In the text editor, the user should be provided with commands to move left or right by units of characters, words, lines, or by the entire area of the text. To specify these commands we need a to be able to recognize these significant locations within the text. Using notations developed by Sufrin [26], we first define the constant *nl* to stand for a new line character and *sp* for a space.

$$sp : CH$$
$$nl : CH$$
$$sp \neq nl$$

Moving left and right by units of a character is specified as follows:

Move_right_char

ΔBox_Text

$text' = text$

$right' = tail(right)$

Move_left_char

ΔBox_Text

$text' = text$

$left' = front(left)$

Move_right_word

ΔBox_Text

$text' = text$

let $d = find\_nt(\{sp, nl\}, right) - 1$

$right' = succ^d \mathbin{;} right$

Move_left_word

ΔBox_Text

$text' = text$

let $d = find\_ld(\{sp, nl\}, left)$

$left' = (1 \mathinner{..} d) \overset{seq}{\lhd} left$

Move_right_line

ΔBox_Text

$text' = text$

let $d = find\_nt(\{nl\}, right) - 1$

$right' = succ^d \mathbin{;} right$

Move_left_line

ΔBox_Text

$text' = text$

let $d = find\_ld(\{sp, nl\}, left)$

$left' = (1 \mathinner{..} d) \lhd left$

Move_top

ΔBox_Text

$text' = text$

$left' = \langle\rangle$

Move_bottom

ΔBox_Text

$text' = text$

$right' = \langle\rangle$

10.3 *Delete*

Delete_right_char

ΔBox_Text

$left' = left$

$right' = tail(right)$

Delete_left_char

ΔBox_Text

$right' = right$

$left' = front(left)$

Delete_right_word

ΔBox_Text

$left' = left$

let $d = find\_nt(\{sp, nl\}, right) - 1$

$right' = succ^d \mathbin{;} right$

Delete_left_word

ΔBox_Text

$right' = right$

$$\textbf{let } d = find\_ld(\{sp, nl\}, left)$$
$$left' = (1 \mathinner{..} d) \lhd left$$

Delete_right_line

ΔBox_Text

$$left' = left$$
$$\textbf{let } d = find\_nt(\{nl\}, right) - 1$$
$$right' = succ^{d} \fatsemi right$$

Delete_left_line

ΔBox_Text

$$right' = right$$
$$\textbf{let } d = find\_ld(\{sp, nl\}, left)$$
$$left' = (1 \mathinner{..} d) \lhd left$$

11. Item Menu Operations

An item menu is used for selecting from a finite number of choices. An item menu may be modeled as an injective sequence of items where each item is itself a sequence of characters.

$$ITEM == \text{seq } CH$$

Item_Menu

$item\_list : \text{iseq } ITEM$
$list\_size : \mathbb{N}$
$highlight : \mathbb{N}$

$list\_size = \#item\_list$
$list\_size >= 1$
$highlight <= list\_Size$
$highlight >= 1$

The item menu has a fixed number of items as specified by *list_size*. The location of the *selection cursor* within the item menu is given by *highlight*. The initial state of the item menu is partially specified as follows:

Init_Item_Menu

$Item\_Menu$

$$Highlight = 1$$

With this simple model, specifying the behavior of the *Prev_Item* and *Next_Item* commands is straight forward.

11.1 *Prev_Item*

Prev_Item

$$\Delta Item\_Menu$$

$$highlight > 1 \Rightarrow highlight' = highlight - 1$$
$$highlight = 1 \Rightarrow highlight' = list\_size$$

If the highlight is at the first item, wrap around to the last item in the list. Otherwise, move the highlight to the previous item in the list.

11.2 *Next_Item*

Next_Item

$$\Delta Item\_Menu$$

$$highlight < list\_size \Rightarrow highlight' = highlight + 1$$
$$highlight = list\_size \Rightarrow highlight' = 1$$

If the highlight is on the last item, wrap around to the first item in the list. Otherwise, move the highlight to the next item in the list.

11.3 *Mneumonic*

If the item list contains a small number of items, unique mneumonic characters may be assigned to each item. If the user types the assigned character, the highlight moves immediately to the specified choice. The mneumonic assignments to each item it an injective function.

Mneumonic_Menu

$$\Delta Item\_Menu$$
$$ch\_item : CH \overset{inj}{\pfun} ITEM$$

$$\mathrm{rng}\ ch\_item = \mathrm{rng}\ item\_list$$
$$\#ch\_item = \#item\_list$$

The invariants say that both *ch_item* and *item_list* map to the same set of items. Entering a mneumonic menu command is defined by the schema

Mneumonic

$$\Delta Mneumonic\_Menu$$

$cmd? : CH$

$cmd? \in \text{dom}\ ch\_item$

$highlight' = (\mu n : \mathbb{N} \mid ch\_item(cmd?) = item\_list(n))$

The mu-expression (μ) gives the unique value of n that maps *item_list* to the same item as *ch_item*(*cmd*?).

11.4 *Abbreviate*

The IBM Common User Access standard specifies a different method of positioning for larger item lists where it is not practical to assign unique mneumonics. When the user types a letter, the selection cursor moves to the next choice starting with that letter. This kind of item selection assumes the item list is in alphabetic order. To specify this operation we use the function *first_match* to return the index of the first item which begins with the specified letter.

Abbreviate

$\Delta Item\_Menu$

$cmd? : CH$

$cmd? \in (\text{rng}\ item\_menu)(\!|\ 1\ |\!)$

$highlight' = first\_match(cmd?, item\_menu)$

11.5 *Select*

The *Select* command simply returns the value of the highlighted item.

Select

$\Xi Item\_Menu$

$select\_value' : \text{seq}\ CH$

$select\_value' = item\_menu(highlight)$

11.6 *Add_item*

A function may be provided to allow the user to add an item to the list.

Append_Item

$\Delta Item\_Menu$

$new\_item? : \text{seq}\,(CH)$

$new\_item? \notin \text{rng}\ item\_list$

$list\_size' = list\_size + 1$

$item\_list' = item\_list \frown new\_item?$

Since this operation would only apply to larger lists that would use the *Abbreviate* item selection technique, our item list must be sorted into ascending order using the generic *Sort_List* schema.

$$Add\_Item \;\widehat{=}\; Append\_Item \wedge Sort\_List[ITEM][item\_list/in?, item\_list'/out!]$$

12. Box Display Operations

When a box is formatted for display output, it is converted into one text area (i.e., a sequence of characters) containing representations of all the box parts.

12.1 *Move*

The move commands are the same as those for moving about within a text area in combination with the commands for moving among box parts.

12.2 *Search*

The search commands provides a means to move directly to the next or previous location in text containing a specified pattern.

search_right

ΔBox_Text

$pattern? : \text{seq } CH$

$right \overset{\cdots}{\hookrightarrow} pattern$

let $d = find\_first(pattern, right)$

$right' = succ^d \; ; \; right$

$text' = text$

search_left

ΔBox_Text

$pattern? : \text{seq } CH$

$left \overset{\cdots}{\hookrightarrow} pattern$

let $d = find\_last(pattern, left)$

$left' = 1 \,..\, d \lhd left$

$text' = text$

CHAPTER 10

Conclusions

10.1 Summary

We presented a method for combining the Z notation with the Box Structure Method to develop a two-stage specification consisting of a black box followed by a state box. In the black box, transition rules are expressed by describing valid stimulus histories for given stimulus-response pairs. In the black box, individual transition rules may be expressed formally or informally, but must be partitioned according to stimulus-response pairs. By developing a top-level black box first, we avoid a premature commitment to specifying the state data until we have first identified the state data requirements.

State data are defined by examining the black box transition rules to discover state data requirements. State space schemas are used to define state data variables as mathematical data types. Operation schemas are used to express state box transition rules.

Using a mathematical notation such as Z to express requirements offers many advantages over using informal text alone. The exercise of developing a mathematical model helps the specifier to develop a better understanding of the application problem. Since formal notations are more precise and concise than informal notations, they provide a better basis for communication and verification.

Although we have not discussed formal verification of the state box in this paper, we have shown a correspondence between the state box view and black box view that should facilitate such methods.

10.2 Future work

The Box Structure Assistant is specified to support creating, editing, and printing system designs using the C Design Language. This specification will be used by a Cleanroom team to code, verify and certify this system. This experiment will provide a realistic appraisal of this specification method.

We plan to expand our use of the Z notation for expressing black box transition rules. As we gain more experience in this technique, we hope to define more expressive relations and functions for making assertions on traces that would be useful for the specification of large complex systems.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Abrial, J. R., "The specification language Z," Oxford Programming Research Group, Technical Report, 1980.

[2] Bartussek, W., and D. L. Parnas, "Using assertions about traces to write abstract specifications for software modules," *Second Conference of the European Cooperation in Informatics*, Springer-Verlag, 1978.

[3] Cobb, R. H., and H. D. Mills, "Engineering software under statistical quality control," *IEEE Software*, pp. 44–56, November 1990.

[4] Cohen, B., Harwood, W. T., Jackson, M. I., *The Specification of Complex Systems*, Addison-Wesley, 1986.

[5] Enderton, H. B., *Elements of Set Theory*, Academic Press, New York, 1977.

[6] Hall, A., "Seven myths of formal methods," *IEEE Software*, pp. 11–19, September 1990.

[7] Hartson, R. H., and D. Hix, "Human-computer interface development concepts and systems," *ACM Computing Surveys*, Vol. 21, No. 1, pp. 5–92, March 1989.

[8] Hayes, I. (editor), *Specification Case Studies*, Prentice-Hall International, London, 1987.

[9] Hoffman, D., "Practical interface specification," *Software: Practice and Experience*, Vol. 19(2), 127–148, February 1989.

[10] IBM Corporation, *Systems Application Architecture, Common User Access: Advanced Interface Design Guide*, SC26-4582, New York, 1989.

[11] IBM Corporation, *Systems Application Architecture, Common User Access: Basic Interface Design Guide*, SC26-4583, New York, 1989.

[12] Ince, D. C., *An Introduction to Discrete Mathematics and Formal System Specification*, Oxford University Press, Oxford, England, UK, 1988.

[13] Jones, C. B., *Systematic Software Development using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[14] Knuth, D. E., *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1984.

[15] Liskov, B. H., Berzins, V., "An Appraisal of Program Specifications", *Software Specification Techniques*, Gehani, N., McGettrick, A. D. (editors), Addison-Wesley, 1986.

[16] Linger, R. C., H. D. Mills, and B. I. Witt, *Structured Programming*, Addison-Wesley, Reading, MA, 1979.

[17] Mills, H. D., "Stepwise refinement and verification in box-structured systems," *Computer*, IEEE, pp. 23–26, June 1988.

[18] Mills, H. D., R. C. Linger, and A. R. Hevner, *Principles of Information System Analysis and Design*, Academic Press, Inc., 1986.

[19] McMorran, M. A., J. E. Nicholls, *Z User Manual*, IBM Technical Report TR12.274, Version 1.0, Hursley Park, Winchester, UK, 1989.

[20] Morgan, C., and B. A. Sufrin, "Specification of the Unix filing system," *Computer*, IEEE, vol. 22, no. 2, pp. 8–24, 1984.

[21] Open Software Foundation, *OSF/Motif Style Guide (revision 1.0)*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[22] Senn, C., *Box Structure BDL Generator: A Clerical Tool to Support Software Development*, Masters Thesis, University of Tennessee, Knoxville, 1989.

[23] Senn, C., C. Trammell, H. Mao, N. Sims, R. Baker, "Specification for the Box Structure BDL Generator," Technical Report, Department of Computer Science, University of Tennessee, Knoxville, TN, 1989.

[24] Spivey, J. M., *The Z Notation: A Reference Manual (Computer Science Series).*, Prentice-Hall Englewood Cliffs, NJ, 1989.

[25] Spivey, J, M., *Understanding Z: A specification language and its formal semantics*, Cambridge University Press, Cambridge, 1988.

[26] Sufrin, B. A., "Formal specification of a display oriented text editor," Gehani, N., and A. D. McGettrick (eds.), *Software Specification Techniques*, Addison-Wesley, 1986.

[27] Sufrin, B. A., and H. Jifeng, "Specification, analysis and refinement of interactive processes," M. Harrison and H. Thimbleby (eds.), *Formal Methods in Human-Computer Interaction*, Cambridge University Press, Cambridge 1990.

[28] Wing, J. M., "A Specifier's Introduction to Formal Methods", *Computer*, IEEE, vol. 23, no. 9, pp. 8–24, September 1990.

APPENDIX

Glossary of the Z Notation

Definitions and declarations

Let x be an identifier and T be a set.

$\widehat{=}$	"is defined to be". E.g., $LHS \mathrel{\widehat{=}} RHS$ defines LHS as syntactically equivalent to RHS.
$==$	Defines an abbreviation (e.g., *ident* $==$ *expression*). The identifier on the left becomes a global constant with the same value and type as the expression on the right.
$x : T$	Declare x as a variable of type T. Values of x are constrained to lie within the set T.
$[A, B]$	Introduce basic types.

Logic

Let P and Q be predicates and let D be a declaration.

true, false	Logical constants.
$\neg P$	Negation: "not P".
$P \wedge Q$	Conjunction: "P and Q".
$P \vee Q$	Disjunction: "P or Q".
$P \Rightarrow Q$	Implication: "P implies Q" or "if P then Q".
$\forall x : T \bullet P$	Universal quantification: 'for all x of type T, P holds'.
$\exists x : T \bullet P$	Existential quantification: 'there exists an x of type T such that P'.
$\exists! x : T \bullet P$	Unique existence: 'there exists exactly one x of type T such that P'.

Sets

Let S, T and X be sets; t, t_k terms; P a predicate and D declarations.

$t \in S$	Set membership: "t is an element of S".
$t \notin S$	$\widehat{=} \neg(t \in S)$.

$\emptyset$ The empty set.

$\{t_1, t_2, \ldots, t_n\}$ The set containing $t_1, t_2, \ldots, t_n$.

$\{x : T \mid P\}$ The set containing exactly those x of type T for which P holds.

$\{D \mid P \bullet t\}$ The set of t's such that, given the declarations D, P holds.

$\{D \bullet t\}$ $\widehat{=} \{D \mid true \bullet t\}$

$(t_1, t_2, \ldots, t_n)$ An ordered n-tuple whose components are $t_1, t_2, \ldots, t_n$.

$T_1 \times T_2 \times \cdots \times T_n$ Cartesian product. The set of all n-tuples $(t_1, t_2, \ldots, t_n)$ where $t_i \in T_i$ for each i with $1 \leq i \leq n$.
$\widehat{=} \{t_1 : T_1; \ldots; t_n : T_n \bullet (t_1, \ldots, t_n)\}$

$first(x, y)$ Projection function to extract the first element from an ordered pair.
$\widehat{=} x$

$second(x, y)$ Projection function to extract the second element from an ordered pair.
$\widehat{=} y$

$\mathbb{P} S$ Power set: the set of all subsets of S.

$\mathbb{P}_1 S$ Non-empty subsets
$== \{s : \mathbb{P} S \mid s \neq \emptyset\}$

$S \cup T$ Set union
$\widehat{=} \{x : X \mid x \in S \vee x \in T\}$.

$S \cap T$ Set intersection
$\widehat{=} \{x : X \mid x \in S \wedge x \in T\}$.

$S \setminus T$ Set difference
$\widehat{=} \{x : X \mid x \in S \wedge x \notin T\}$.

$\bigcup SS$ Generalized union
$\widehat{=} \{x : X \mid \exists S : SS \bullet x \in S\}$

$\#S$ The cardinality (number of distinct elements) of a finite set.

$\mathbb{F} S$ The set of all finite subsets of S.
$\widehat{=} \{S : \mathbb{P} X \mid \exists n : \mathbb{N} \bullet \exists f : 1..n \rightarrow S \bullet \mathrm{rng}\, f = S\}$

$\mathbb{F}_1 S$ The non-empty finite subsets of S.
$\widehat{=} \mathbb{F} S \setminus \{\emptyset\}$

Numbers

$\mathbb{N}$	The set of natural numbers (non-negative integers).
$\mathbb{N}_1$	The set of strictly positive integers. $\widehat{=} \mathbb{N} \setminus \{0\}$
$\mathcal{Z}$	The set of integers.
$m..n$	The set of integers between m and n inclusive: $\widehat{=} \{k : \mathcal{Z} \mid m \leq k \leq n\}$
$min\ S$	The smallest number in a set of numbers.
$max\ S$	The largest number in a set of numbers.
$\mu x : T \mid P$	The unique x of type T which satisfies P.

Relations

A relation is a set of ordered pairs. Since a relation is a set, operators defined for sets can be used on relations. Let X, Y, and Z be sets; $x : X$; $y : Y$; and $R : X \leftrightarrow Y$.

$X \leftrightarrow Y$	The set of binary *relations* between X to Y. Each member relation is a subset of $X \times Y$. $\widehat{=} \mathbb{P}(X \times Y)$.
$x\ R\ y$	x is related to y by R: $\widehat{=} (x, y) \in R$.
$x \mapsto y$	$\widehat{=} (x, y)$.
dom R	The domain of a relation: $\widehat{=} \{x : X; y : Y \mid (x \mapsto y) \in R \bullet x\}$.
rng R	The range of a relation: $\widehat{=} \{x : X; y : Y \mid (x \mapsto y) \in R \bullet y\}$.
$R_1 \fatsemi R_2$	Forward relational composition: given $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$, $\widehat{=} \{x : X; z : Z \mid (\exists y : Y \bullet x R_1 y \wedge y R_2 z)\}$.
$R_1 \circ R_2$	Relational composition: $\widehat{=} R_2 \fatsemi R_1$.
R^{-1}	Inverse of relation R: $\widehat{=} \{y : Y; x : X \mid x R y\}$.

$id\, X$ — Identify function on the set X:
$\widehat{=} \{x : X \bullet (x, x)\}$.

R^k — The relation $R : X \leftrightarrow X$ is composed with itself through k iterations:
$R^0 \widehat{=} id\, X, R^{k+1} \widehat{=} R \circ R^k$.

R^* — Reflexive transitive closure:
$\widehat{=} \bigcup \{n : \mathbb{N} \bullet R^n\}$.

R^+ — Non-reflexive transitive closure:
$\widehat{=} \bigcup \{n : \mathbb{N}_1 \bullet R^n\}$.

$S \lhd R$ — Domain restriction to S. Given $S : \mathbb{P}\, X$,
$\widehat{=} \{x : X; y : Y \mid x \in S \wedge x R y\}$.

$S \lhd\!\!\!\!- R$ — Domain subtraction of S. Given $S : \mathbb{P}\, X$,
$\widehat{=} (X \setminus S) \lhd R$.

$R \rhd T$ — Range restriction to T. Given $T : \mathbb{P}\, Y$,
$\widehat{=} \{x : X; y : Y \mid y \in T \wedge x R y\}$.

$R \rhd\!\!\!\!- T$ — Range subtraction of T. Given $T : \mathbb{P}\, Y$,
$\widehat{=} R \rhd (Y \setminus T)$.

$R(\!|S|\!)$ — Relational image given $S : \mathbb{P}\, X$
$\widehat{=} \mathrm{rng}\,(S \lhd R)$.

Functions

$X \nrightarrow Y$ — The set of partial functions from X to Y.
$\widehat{=} \{f : X \leftrightarrow Y \mid (\forall x : \mathrm{dom}\, f \bullet (\exists! y : Y \bullet x f y))\}$

$X \rightarrow Y$ — The set of total functions from X to Y.
$\widehat{=} \{f : X \nrightarrow Y \mid \mathrm{dom}\, f = X\}$

$X \overset{inj}{\nrightarrow} Y$ — The set of partial injections (one-to-one) from X to Y.
$\widehat{=} \{f : X \nrightarrow Y \mid (\forall y : \mathrm{rng}\, f \bullet (\exists! x : X \bullet x f y))\}$

$X \overset{inj}{\rightarrow} Y$ — The set of total injections from X to Y. An *injection* is a function whose inverse is also a function.
$\widehat{=} \{f : X \overset{inj}{\nrightarrow} Y \mid \mathrm{dom}\, f = X\}$

$X \twoheadrightarrow Y$ — The set of partial surjections from X to Y. The range of a surjective function is the entire set from which the second elements of its pairs are taken.
$\widehat{=} \{f : X \nrightarrow Y \mid \mathrm{rng}\, f = Y\}$

$X \twoheadrightarrow Y$ — The set of total surjections from X to Y.
$\widehat{=} (X \twoheadrightarrow Y) \cap (X \rightarrow Y)$

$X \overset{inj}{\rightarrow} Y$ — The set of bijections from X to Y. Bijections are both injective and surjective.
$\widehat{=} (X \rightarrow Y) \cap (X \overset{inj}{\rightarrow} Y)$

$X \nrightarrow Y$ — The set of partial finite functions from X to Y. These are partial functions whose domain is a finite subset of X.
$\widehat{=} \{f : X \nrightarrow Y \mid \mathrm{dom}\, f \in \mathbb{F}\, X\}$

$X \overset{inj}{\nrightarrow} Y$ — The set of partial finite injections from X to Y. These are partial finite functions that are also injections.
$\widehat{=} (X \nrightarrow Y) \cap (X \overset{inj}{\nrightarrow} Y)$

$\lambda x : T \mid P \bullet t$ — Lambda-notation.
$\widehat{=} \{x : T \mid P \bullet x \mapsto t\}$

$f(x)$ — Function f applied to argument x.
$\widehat{=} \mu y : Y \mid x f y$

succ — Function which returns the successor number of its argument.
$\widehat{=} \lambda n : \mathbb{N}_1 \bullet n + 1$

pred — Function which returns its predecessor number of its argument.
$\widehat{=} \lambda n : \mathbb{N}_1 \bullet n - 1$

Sequences

$\mathrm{seq}\, X$ — $\mathrm{seq}\, X$ is the set of finite sequences whose elements are drawn from X.
$\widehat{=} \{A : \mathbb{N}_1 \nrightarrow X \mid (\exists n : \mathbb{N} \bullet \mathrm{dom}\, A = 1 \,..\, n)\}$

$\mathrm{iseq}\, X$ — $\mathrm{iseq}\, X$ is the set of finite injective sequences whose elements are drawn from X. These are sequences which do not have repetitions.
$\widehat{=} \mathrm{seq}\, X \cap (\mathbb{N} \overset{inj}{\nrightarrow} X)$

$s \frown t$ — The *concatenation* of sequences s and t.

$\#A$ — The length of sequence A.

$\langle\rangle$ — The empty sequence.
$\widehat{=} \emptyset$

$\langle x_1, \ldots, x_n \rangle$ — The sequence containing $x_1, \ldots, x_n$.
$\widehat{=} \{1 \mapsto x_1, \ldots, n \mapsto x_n\}$

head A — The first element of a non-empty sequence.
$\widehat{=} A \neq \langle\rangle \Rightarrow head\ A = A(1)$

last A — The final element of a non-empty sequence.
$\widehat{=} A \neq \langle\rangle \Rightarrow last\ A = A(\#A)$

tail A — A new sequence containing all of the elements of A except for the first.
$\widehat{=} tail(\langle x\rangle \frown A) = A$

front A — A new sequence containing all of the elements of A except for the last.
$\widehat{=} front(A \frown \langle x\rangle) = A$

disjoint *AS* — Pairwise disjoint: given $AS : \mathrm{seq}(\mathbb{P}\, X)$,
$\widehat{=} (\forall i, j : \mathrm{dom}\, AS \bullet i \neq j \Rightarrow AS(i) \cap AS(j) = \emptyset)$

AS **partitions** *S* — The indexed family of sets *AS partitions* the set S if they are disjoint and the union of all sets $AS(i)$ is S.
$\widehat{=}$ **disjoint** $AS \wedge \bigcup \mathrm{rng}\, AS = S$

squash F — If F is a function whose domain is a set of natural numbers, then *squash* F is the sequence formed by arranging the range elements of F in ascending order of their corresponding domain values. For example, if $F = \{9 \mapsto Z, 4 \mapsto X, 6 \mapsto Y\}$ then *squash* $F = \langle X, Y, Z\rangle$.

$A \overset{seq}{\rhd} T$ — Restrict the range of sequence A to the set T. The result is a sequence restricted to just those elements that are in T, but in the same order as in A.
$\widehat{=} squash(A \rhd T)$

$T \overset{seq}{\lhd} A$ — Restrict the domain of the sequence A to the numbers in T.
$\widehat{=} squash(T \lhd A)$

VITA

Daniel Thomas Fetzer was born in Elizabethton, Tennessee on May 29, 1954. He attended elementary schools in that town and graduated from Elizabethton High School in 1972. In the Fall of 1972, he began studies at East Tennessee State University majoring first in music, and later in biology. In the Fall of 1975, Dan left school temporarily to pursue a career as a professional musician. He resumed his education in 1979 and received a Bachelor of Science degree in Computer Science in 1980.

Upon graduation, Dan accepted a *Programmer/Analyst* position with Oak Ridge Associated Universities in Oak Ridge, Tennessee. During his tenure with this organization, he also held positions of *Lead Programmer/Analyst, Applications Programming Supervisor,* and *Systems Analyst.* Dan is currently employed as a *Computing Specialist* by Martin Marietta Energy Systems, Oak Ridge, Tennessee.

Dan received the Master of Science degree in Computer Science in May 1992. He is a member of Association of Computing Machinery and the IEEE Computer Society.