A09 2[illegible]

LEVEL III

12

NOSC

NOSC TD 366

NOSC TD 366

Technical Document 366

AD A091272

THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK

Volume III: A Detailed Example in the use of HDM

June 1979

Prepared for

Naval Ocean Systems Center

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

80 11 04 044

ADMINISTRATIVE INFORMATION

The HDM Project was funded under Navy Element 62721N. The work leading to this publication was the result of NOSC Contract N00123-76-C-0195 with SRI International. The principal researchers were Dr. Karl Levitt (Volume III), Mr. Lawrence Robinson (Volume I), and Dr. Brad A. Silverberg (Volume II). The Navy Technical Monitor for the work performed under contract was W. Linwood Sutton, NOSC Code 8324.

Reviewed by
J. B. Balistrieri, Acting Head
C^3I Facilities Engineering &
Development Division

Under authority of
V. J. Monteleon, Acting Head
Command, Control, Communications
and Intelligence Systems Department

(18) NOSC (19) TD-366-VOL-3

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NOSC Technical Document 366 — 2. GOVT ACCESSION NO. AD-A091 272	3. RECIPIENT'S CATALOG NUMBER Technical document
4. TITLE (and Subtitle) (6) THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK. Volume III. A Detailed Example in the Use of HDM.	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) W. Linwood/Sutton ~~[illegible]~~ Karl N./Levitt	8. CONTRACT OR GRANT NUMBER(s) (15) N00123-76-C-0195
9. PERFORMING ORGANIZATION NAME AND ADDRESS SRI International 333 Ravenswood Avenue Menlo Park, CA 94025 Lawrence/Robinson Bernard A./Silverberg	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62721N
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Ocean Systems Center San Diego, CA 92152 (11)	12. REPORT DATE June 1979
	13. NUMBER OF PAGES 172
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Ocean Systems Center San Diego, CA 92152 (12) 175	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

abstract machines, abstraction, formal specification, hierarchical structure, Hierarchical Development Methodology (HDM), modules, software development process, software methodology.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

HDM (the SRI Hierarchical Development Methodology) is an approach to software development that attempts to structure the overall development process by providing a unified framework that addresses most aspects of system development. Volume III of the HDM Handbook illustrates the application of HDM to the development of a non-trivial system. The example system is developed step-by-step, the languages of HDM are presented, and the decisions actually made during system development are described.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

410 241

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SRI International

THE HDM HANDBOOK

Volume III: A Detailed Example in the Use of HDM

Deliverable A006

SRI Project 4828
Contract N00123-76-C-0195

June 1979

By: Karl N. Levitt, Program Manager
Lawrence Robinson, Computer Scientist
Brad A. Silverberg, Computer Scientist

Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Naval Ocean Systems Center
San Diego, California 95152

Attention: W. Linwood Sutton, Contract Monitor

CONTENTS

Foreword

This is the third and final volume of the HDM Handbook. The first volume covers the concepts and general philosophy of HDM. The second volume discusses HDM's languages and tools. This volume presents in detail an example of the use of HDM.

I FOLLOWING THE EXAMPLE

In carrying out the example, we follow the seven stages of development. Many pages are required for the specifications, representations, and implementations, and for the supporting explanations and diagrams. We anticipate the reader's questioning the need for such a lengthy discussion of an example that is certainly not a large system. However, even with a moderate size system (such as ours) there are a multitude of decisions that need to be made during the steps of design, specification, and implementation. Using conventional design techniques, many of these decisions are made informally, often without even realizing that a decision has been made. Instead, in HDM we make each decision explicit. As a result, the discussion of these decisions gets rather lengthy, though we contend that these decisions must be confronted by the designer (explicitly or not) in any case. We recognize that for an example of this size, many programmers would not need the complete documentation offered by HDM in that they could retain the decisions in their heads. Once a critical size is reached, however, the number of decisions to be made becomes unwieldy unless a systematic approach (such as the methodology of HDM) is adopted. We ask the indulgence of those programmers and suggest that they extrapolate in their minds to a larger system which might be more taxing.

The reader who carefully follows the details will gain an understanding of how HDM encourages the developer to contemplate and formulate decisions. In addition, he will be exposed to most of the features of the languages of HDM. We recognize that many readers will not be able to devote such careful effort to the example. To those we recommend just a study of any two levels -- preferably the top two -- of the example, and that they follow the development of these levels through all stages. An adequate introduction to the application of HDM will be attained by this process.

We confess to the reader that the example occasionally departs from reality to illustrate important features of HDM. For example, our implementation of a file system is not realistic; it has been chosen to illustrate the use of a module in more than one machine. In addition, the design is somewhat more general than required. This is done to include mechanisms appropriate to a large class of searching problems. A few extra modules have been included to enable verification of the implementations, which is not carried out here. In addition, we have included more modules than necessary, in order to yield near trivial representations and implementations. Some designers might accept more complicated implementations at the benefit of fewer modules and representations.

In Chapter 2, we present the high-level details of our example system. In Chapter 3, we describe the modules of the extreme machines, and in Chapter 4 we describe the modules of the intermediate machines. In Chapter 5 we present the SPECIAL specifications for these system's modules, and in Chapter 6 we present the machine representations. We conclude this volume with Chapter 7, which contains a presentation of the module implementations.

II CONCEPTUALIZATION: STAGE 1

In this stage, the overall intent of the system is described in a sufficiently abstract manner to leave out detailed design decisions. Since we have yet to develop a formal language for conceptualization, our descriptions will be informal.

At the user level, the system is to provide the following functional capabilities:

* Provide a simple file system. The file system is to manage a collection of files (word sequences). No explicit constraint is imposed on the length of word that can be stored in a file.

* Provide a facility to enable the establishment of a frequency count for the words of a user-designated file. It is assumed that the user will wish to print the set of words (each word once) in the designated file, together with the frequency count for each word. The words are to be printed according to their order in the file. The printer is outside the scope of this example, but the words and their frequency counts are to be stored in a form that is suitable for printout. In this discussion, we will refer to the set of words and their frequency counts as the <u>histogram</u> for the designated file.

* For purposes of computing the histogram, words in the designated file are truncated to some length that is fixed at system initialization. Thus, if the truncation length is three, the words "patch" and "pat" are both treated as "pat". The truncation feature can be used for a given file to count all words that have a particular prefix.

Two considerations we impose on the design relate to modifiability and performance:

* With relatively minor extensions to the system, the user will be able to compute the histogram to facilitate a printout order based on other criteria, e.g., the number of occurrences of a word, the length of a word, or alphabetical order.

* Wherever possible, the system performance is to be optimized in favor of "time", possibly at the expense of "space".

The first consideration will influence the operations provided by the level immediately below the user-interface. It must provide capabilities over and above those being utilized by the current user-interface, so that the user-interface can be altered without affecting the lower levels.

Finally, in this stage we need to decide upon the primitive machine for our example. Sometimes, we choose the primitive machine to be a given piece of hardware. More often, though, we choose a suitably idealized primitive machine, yet one sufficiently simple to be easily implemented on any hardware. For our example, we follow the latter route, and assume:

* The primitive machine is to provide two kinds of arrays, one kind for characters and the other for integers.

Let us illustrate some of the decisions that are indicated above.

Figure II-1a depicts a file containing six words. The histogram for this file, as it might appear on the printer, is shown in Figure II-1b. Note that the words are printed in the order of their first appearance in the file and that for a word containing more than three characters, only the first three characters are considered in producing the histogram.

If the print order is desired according to word frequency, then the printer output is as depicted in Figure II-2. The conceptualization does not prescribe the print order for a set of words all with the same frequency counts. We have arbitrarily depicted the order among such words to be according to first appearance in the file.

The reader should note that much is (intentionally) left unspecified in the above conceptualization; for example, nothing has been said about:

* What is a word.
* How words are stored in the individual files.
* What operations are provided to the user.
* How the words of the histogram are stored in order to facilitate their printout according to the desired criteria.
* How the computations are to utilize the given truncation length.

We will observe how these, and other issues, are confronted by

Figure II-1: Histogram for a File -- Ordered by First Appearance

1	DOG
2	AT
3	A
4	PATCH
5	PAT
6	AT

(a)

HISTOGRAM FOR FILE X

ORDER	WORD	FREQUENCY
1	DOG	1
2	AT	2
3	A	1
4	PAT	2

PRINTED HISTOGRAM, TRUNCATION LENGTH = 3

(b)

Figure II-2: Histogram for a File -- Ordered by Word Frequency

HISTOGRAM FOR FILE X

ORDER	WORD	FREQUENCY
1	AT	2
2	PAT	2
3	DOG	1
4	A	1

proceeding to the next stages.

We must keep in mind as we progress that design is an iterative process. Oftentimes, the design process did not go as smoothly (nor as linearly) as might be suggested by the following chapters. Sometimes we had to backtrack and re-think certain decisions. This is inevitable for a non-trivial system design implementation.

Another consideration that must be kept in mind is how the specification language shapes the design itself, and how intuitive ideas are expressed in the specification language. In particular, we want to emphasize that conceptual "abstract data types" are manifested in SPECIAL as designator types. That is, whenever we want an abstract type, we typically specify it in SPECIAL as a designator type. Conversely, whenever we see a designator type in a SPECIAL specification, we typically have in mind a corresponding abstract type. Thus, when we say that a given module supports designator type "x", we mean that it supports the corresponding conceptual abstract type.

III DEFINITION OF EXTREME MACHINES: STAGE 2

In this stage we organize the extreme machines (the user-interface and primitive machines) into modules and list the operations and data structures for each of these machines. We focus initial attention on these two machines because of their criticality in the overall system design. The user-interface is the only part of the system the user interacts with directly. The primitive machine provides the foundation on which the system rests. Thus, decisions relating to these two machines are particularly significant -- these machines involve more than just the development team. On the other hand, there is more freedom in the selection of the intermediate machines (see next chapter), and decisions underlying their design involve only the design team. In HDM we postpone their consideration until later.

A. Presenting the Module Functions

At each stage of development, HDM is used to record the decisions made at that stage. For Stage 2, the decisions relate to the following:

* The decomposition of the user-interface and primitive machines into modules.

* The functions, parameters, and designator and scalar types for each extreme machine module.

(Note that these are refinements of what we indicated in Volume II as the product of this stage.) It is convenient at this stage to divide the data structures into two classes: those that are not subject to modification (parameters), and those whose value can potentially change (V-functions). In this stage, we do not specify the behavior of functions and parameters; rather, we just record their headers, which include declarations for the formal arguments and results. Exported types (i.e., designator and scalar types) are also distinguished in this stage.

In principle, the documentation for some stages (in particular, those that yield specifications, representations and implementations) should be adequate for conveying the decisions made in that stage.

However, it is clear that an accompanying informal description is a significant help to a reader. For this stage (and Stage 3), the informal description is particularly valuable in conveying decisions that are in the designer's mind as he writes down the module organization and functions, parameters and types, but are not completely captured in the notation of the specification language. Hence, we typically begin each module's specification with an English description of the decisions that are made there and often allude to the many decisions yet to be made.

It is emphasized that some of the decisions of this early stage are subject to change as the system develops. We have attempted to structure the discussion to reflect our thinking as we developed the system, not as an after-the-fact description of the system. Thus, missing from our initial description of the primitive machine are a few auxiliary functions and modules, whose need were not apparent until later stages.

B. Definition of The User Interface

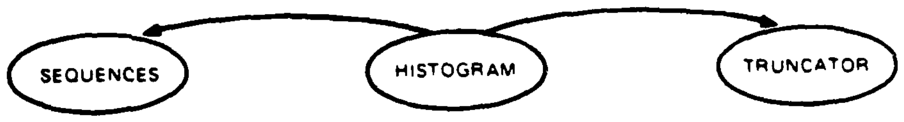
Recall that the intent of our system is to provide:

* A simple file system for word storage and retrieval.
* The capability of computing frequency counts of words in a designated word file (sequence). For purposes of computing the frequency counts, the words of the file are viewed as being truncated to some given length.
* The capability of organizing the distinct words of a sequence and their respective frequency counts to facilitate their printout. The order of printout is to be by first appearance in the designated sequence, but other orders, selectable by the users, should be achievable with minor modifications to the system.

Notice how we were able to partition the intent of the system into three distinct, separable concerns. This naturally determines the modularization of the user interface: one module for each concern.

The "sequences" module embodies the concept of word files; "histogram" provides the mechanism for creating and storing a histogram

Figure III-1: Modular Decomposition of the User Interface



for a given sequence; "truncator" encapsulates the decision concerning the truncation of words for computing the histogram. The facility for truncation was placed in a module separate from that for computing the histogram in order to emphasize that a change in the truncation length has no bearing on the details of the "histogram" module.

The intermodule referencing is shown in Figure III-1.

Let us consider the three modules in turn.

1. Sequences

The "sequences" module is intended to resemble an extremely simple file system. The major decisions are the following:

* Each word file is viewed as an abstract object called a "sequence". Correspondingly, we provide a designator type for sequences, called the "seq" designator type.

* Sequences can be dynamically created.

* Sequences are composed of "words", i.e., vectors of characters.

* A sequence is grown by appending words, one at a time, to its *end*.

* For a given sequence, operations are provided to indicate its length, to retrieve a word at a given position in the sequence, and to interchange the words in two given positions.

The following operations and data structures are provided for the "seq" designator type:

string(seq n; INTEGER j) -> word w -- a primitive visible V-function that returns the j-th word w in sequence n. "Word" is a named type that was informally defined above to have as values all non-null vectors of characters; it is precisely defined later. As the only primitive V-function, "string" captures the state of each sequence in the system.

seqlen(seq n) -> INTEGER v -- a derived visible V-function that returns the current length of sequence n. As we will observe, the value of seqlen(n) can be derived from the primitive V-function "string".

create_seq() -> seq n -- an OV-function that creates a new sequence, initializes it, and assigns a designator to it.

clear_seq(seq n) -- an O-function that clears a designated sequence.

append(seq n; word w) -- an O-function that appends word w to the end of sequence n.

swap_seq(seq n; INTEGER i, j) -- an O-function that causes the words in positions i and j to be exchanged. The need for this operation is not intended to be apparent at this point. The module "sequences", besides providing the abstraction of a file system for users of the system, also appears in an intermediate machine to implement other modules. The operation "swap_seq" is required to carry out this implementation, but need not be made available at the user-interface.

Thus we see that this module somewhat resembles a sequential file system, where the primitive element (record) in an individual file is a "word". In our module, random access retrieval and exchanging of elements is permitted, though words may be added only to the end.

Clearly, we have only characterized the <u>syntax</u> of the file system interface. Aspects of the <u>semantics</u> we have not indicated are:

* The state of an initialized sequence
* The state of a "cleared" sequence
* The maximum word size that can be accommodated
* The maximum number of words that can be accommodated
* How any operation is implemented
* What effect the O- and OV-functions have on the primitive V-functions

Decisions relating to these issues are probably in our minds at this point, but are not fully formulated until later stages.

2. Truncator

The histogram for a sequence is computed with respect to the truncated words in the sequence. Truncator is a useful deployment of modularity to separate the decisions regarding the computation and retrieval of the histogram from those underlying the truncation of words prior to their processing.

The major decision being made here is that the length to which words are truncated is embodied in the initial state of the "truncator" module. The "truncator" module provides the integer-valued parameter "maxlength", which is the truncation length for words.

The V-function "truncation(word w) -> truncated_word tw" returns for a given word w of length greater than maxlength a new word tw that contains the first maxlength characters of w; if the length of w is less than or equal to maxlength, then tw is equal to w. Note that tw is declared to be of type "truncated_word" to emphasize that the range of "truncation" is limited to those words whose length does not exceed "maxlength".

As we will observe, the function "truncation" is needed both to specify the user-interface and to implement an operation of "histogram". However, a user of the system need not have access to "truncation".

3. Histogram

Briefly, two major decisions are made at this point. The first is to provide an operation that will compute the frequency count for each distinct word in a designated sequence. In carrying out the computation, all characters in a given word beyond the truncation length "maxlength" are ignored.

The second is that the histogram is to be represented by two tables, one to hold the set of words in the sequence (in truncated form) and the other to hold the frequency count of each word.

The following functions are selected in connection with the above decisions:

getword(INTEGER j) -> truncated_word tw -- a primitive visible V-function that, in effect, is the table storing the histogram's truncated words.

howmany(INTEGER j) -> INTEGER i -- a primitive visible V-function that, in effect, is the table storing the frequency counts.

histlen() -> INTEGER v -- a derived, visible V-function that returns the number of words stored in the "getword" table (i.e., the number of distinct words after truncation in the designated sequence).

hist(seq n) -- an O-function that is invoked to form the histogram for sequence n.

clear_hist() -- an O-function that clears the "getword" and "howmany" tables.

Note that there are numerous decisions concerning the histogram module that are not formulated here. For example, we have not determined the order the words are stored in the "getword" table, the resource limitations of the histogram module, nor the algorithm for computing the frequency count of each word in the designated sequence.

4. HSL description of the user-interface

HSL (Hierarchy Specification Language) is a simple language for expressing certain properties of abstract machines, and modules, and their organization into a hierarchy. It is convenient to describe the language by illustrating the HSL description of the user-interface.

```
INTERFACE
   (user-interface
      (sequences WITHOUT swap_seq)
      (histogram)
      (truncator WITHOUT truncation))
```

In general, an HSL description provides the following information about an abstract machine:

* The name given to the machine, in this case "user-interface"

* The modules that comprise the machine

* Any functions, parameters, types or designators of a module that are not to be available to the next higher level machine M, i.e., those that cannot be referenced in a program that implements a module of M. For each module, such unavailable functions are listed following the reserved word WITHOUT.

This completes the discussion of the user-interface. Now let us jump to the primitive machine.

C. Definition of The Primitive Machine

We emphasize here again that the primitive machine was somewhat arbitrarily chosen. We wanted a machine that was low-level enough that it could easily be implemented on any hardware, yet was idealized enough that we would not be tied to the details of any particular piece of hardware. As a result, we chose one that provided character arrays, integer arrays, and their respective designators as the primitive machine. In addition, for simplicity we assume certain operations discussed below are also provided by the primitive machine.

In the conceptualization stage it was decided that all of the abstract data structures would ultimately be implemented in terms of character arrays and integer arrays. Character arrays would be used to hold the characters of which words are composed, and integer arrays to hold the frequency counts of words in the histogram, in addition to other integer values whose need is perhaps not yet apparent. Thus, the primitive machine contains the modules: "intarrays" and "chararrays".

Since these modules are quite similar, the design decisions that underlie the choice of functions, parameters and designator types for both modules can be presented in a single discussion.

- A designator type is associated with each of the two modules. Each array is associated with a unique designator.

- There is no mixture of types in an array, i.e., an array can store either characters or integers. Note that this decision was not based on a limitation of SPECIAL which does allow the type of arguments and values to be a union of two or more types.

- All integer arrays are of the same fixed length.

- All character arrays are of the same fixed length.

- Arrays can be dynamically created. The number of arrays that can be created, however, is limited by the resources of a more primitive machine that realizes "chararrays" and "intarrays".

- The elements of the arrays can be randomly accessed for storage and retrieval.

Now let us consider each of the modules in turn.

1. **Intarrays**

This module provides the following:

intarray -- a designator type whose values denote integer arrays.

leni -- an integer parameter that is the length of each integer array.

getint(intarray m; INTEGER j) -> INTEGER v -- a primitive visible V-function that returns the integer stored at index j of integer array m.

create_intarray() -> intarray m -- an OV-function that associates designator m with a newly created and initialized integer array.

change_int(intarray m; INTEGER j, v) -- an O-function that causes integer v to be stored in position j of integer array m, independent of the value previously stored there.

2. **Chararrays**

This module provides analogous operations for character arrays:

chararray -- a designator type whose values are names of character arrays.

lenc -- an integer parameter that is the length of each character array.

getchar(chararray n; INTEGER j) -> CHAR c -- a primitive visible V-function that returns the character in the j-th position of character array n.

create_chararray() -> chararray n -- an OV-function that associates designator n with a newly created and initialized character array.

change_char(chararray n; INTEGER j; CHAR c) -- an O-function that causes character c to be stored in position j of character array n.

In addition, "chararrays" provides an integer parameter, "maxchararrays", that is the number of chararrays the module can support.

3. Summary

It should be clear that there are numerous design decisions not divulged in this stage, for example, the state of a newly initialized array. These are addressed in Stage 4.

The choice of arrays as the primary primitive storage mechanism is quite natural. Most programming languages provide an array mechanism similar to that of these two modules, although there are differences with respect to dynamic vs. compile-time creation of arrays. Some readers might object to our assuming the existence of designators as primitive entities. Some computers provide descriptors or capabilities which exhibit most of the properties of designators. However, in the absence of such protected names, a new primitive machine, situated below the current machine (composed of "intarrays" and "chararrays") could be defined. This new machine could provide integer arrays and character arrays but named by, say, integers. The representation to this new machine would map the intarray and chararray types to integers. (As we have seen, a designator type can be represented by any set of distinguishable items, e.g., integers, reals, characters, another designator type, or a constructed type.) Thus, the protection offered by the use of designators would still be available to all machines except the lowest level.

Now that we have defined the extreme machines of the system, we will define the intermediate machines proceeding downward.

IV DEFINITION OF INTERMEDIATE MACHINES: STAGE 3

In general, the number of intermediate levels is inversely related to the "distance" in abstraction between the user interface and the primitive machine. If the user interface supports extremely abstract entities (relative to the primitive machine), the number of intermediate levels will typically be large.

In our example system, the distance in abstraction is not unduly large; hence, the number of intermediate levels is also moderate (as it turned out, there are six). We proceed in a top-down manner, starting with the user interface and working toward the primitive machine. We have tried to keep the size of each step small. Small steps mean that the difference in complexity between adjoining levels is also small; hence, each level can be easily understood in terms of the abstractions provided by the next lower level. Putting the small steps together gives us a large step -- from the user interface to the primitive machine.

The significant abstractions of the user interface (level6) are the sequence data type and the "hist" operation, which constructs two tables (for words and frequency counts). The next level (level5) provides the same data abstractions as the user interface; its use is to decompose "hist" into more primitive operations. Thus, level5 is used mostly to support procedure abstraction. The remaining levels of the system are used to provide successively more primitive representations for the abstract types (and their operations) of the respective next higher level. Thus, they are used mostly to support data abstraction.

Figure IV-1 depicts each of the abstract machines of the system, their decomposition into modules and the dependency order of the modules in each machines.

Table 1[1] gives the decomposition in HSL notation, as expected by the HDM tools that deals with interfaces and hierarchy structures.

[1]The tables are included in Appendix A of this volume.

Figure IV-1: Decomposition of System in Machines and Modules

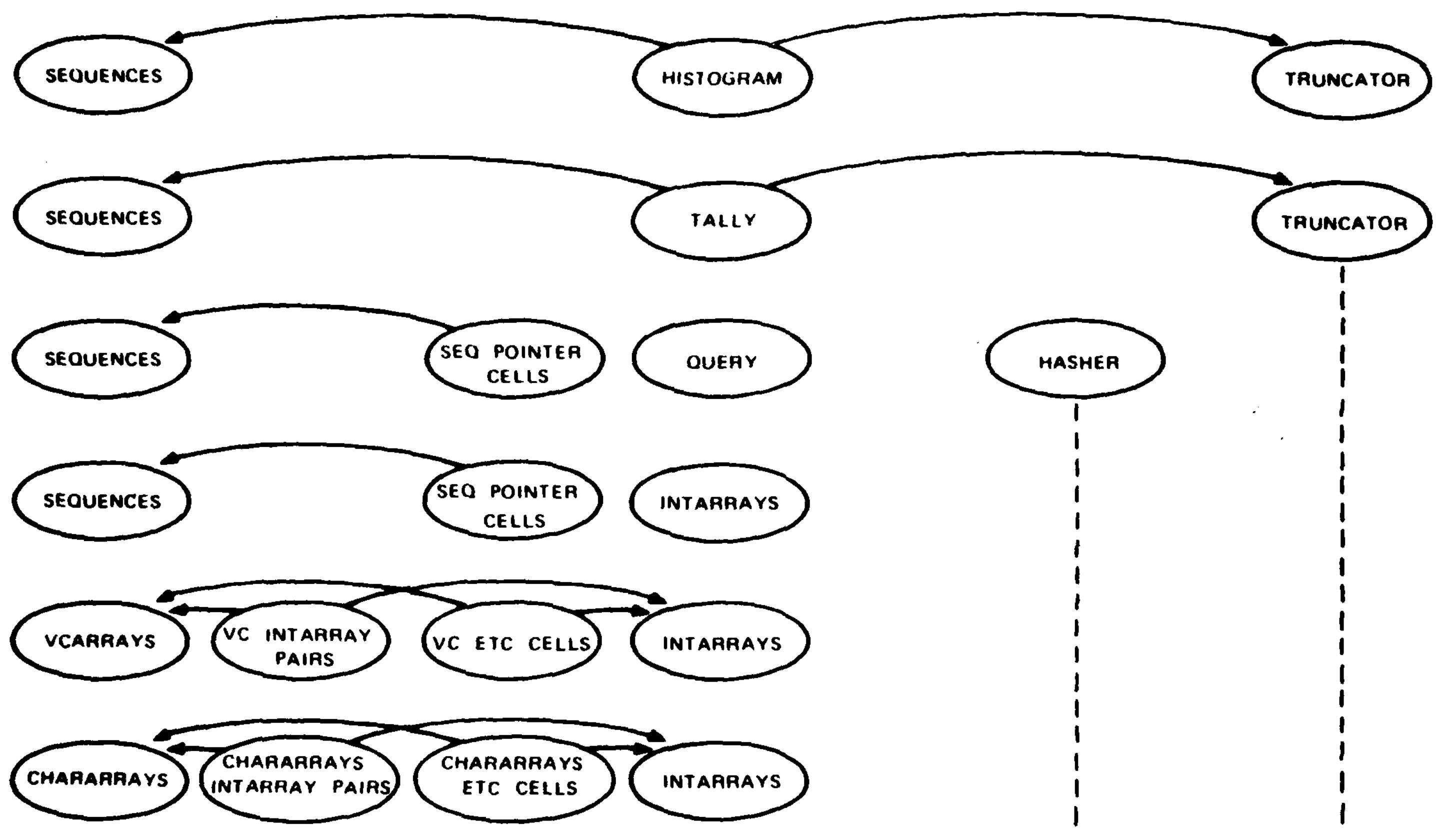


Table 2 lists the functions, module parameters, and designator types of each module. Note for V-functions we employ the following abbreviations: "V" for visible, "H" for hidden, "P" for primitive, and "D" for derived. Also note these headers define only the syntax of each module's functions. We leave the specification of semantics to Stage 4.

A. Level5

The purpose of this machine is to provide facilities for the implementation of "histogram". This is primarily accomplished by the module "tally". While to the user of "histogram" it appears that the two tables (for words and frequency counts) are formed instantaneously, tally builds corresponding tables incrementally with the O-function "insert_or_increment". If the next word in the designated sequence is already in the word-table (as defined by the V-function "t_retrieve"), then "insert_or_increment" increments the count for that word in "t_howmany" by one; otherwise, it adds the word to "t_retrieve" and sets its count in "t_howmany" to one. Finally, "insert_or_increment" also advances the file pointer for the sequence. Our intention is for t_howmany(j) to contain the word count for the word stored in t_retrieve(j).

Note that there is essentially no jump in data abstraction between level6 and level5; "getword" corresponds to "t_retrieve", and "howmany" corresponds to "t_howmany". The main difference between the levels is in procedure abstraction. A few other decisions embodied in the selection of operations and data structures of "tally" are noted.

The hidden, primitive V-function

t_sequence() -> seq s

is used to denote the word sequence currently being processed. When the processing of this sequence is complete, "tally" will generally be reset before another sequence s' is selected for processing. However, without resetting, another sequence s' can be selected by invoking the O-function

t_initialize(s').

Thus, the histograms for two (or more) composed sequences s,s' can be formed. This generality in "tally" is not needed by "histogram" as described, but does not deleteriously impact the efficiency, and might be useful if the conceptualization underlying "histogram" is changed.

The hidden, primitive V-function

t_pointer() -> INTEGER v

indicates the next word to be processed in the designated sequence. Thus, t_pointer() serves as a file pointer for this sequence. A reader might believe that it should be visible, i.e., available to the implementation of "hist" for determining when the end-of-file is reached. As will be noted in the next chapter, we decided to report the "end-of-file" condition as an exception to "insert-or-increment".

Once the processing of a sequence is complete, it might be desired to rearrange the word-table and frequency-table in order to facilitate the printing of the words in a different order, e.g., by length or in alphabetical order. The O-function

swap_tally(INTEGER i, j)

could be used by the implementation of "hist" for this purpose. Thus, we have made the decision in forming the tables to first get all of the words and their associated frequency counts into the tables, and later to arrange the order of the tables.

Note that level5 also contains appearances of the modules "sequences" and "truncator". The role of the former is to give "tally" access to the designated sequence, as passed down in the implementation of "hist". For the latter, the parameter "maxlength" of "truncator" is used to convert a word of the designated sequence into a "truncated_word".

Now let us proceed to level4, which implements tally.

B. Level4

The purpose of this machine -- containing the four modules "sequences", "query", "hasher", and "truncator" -- is to implement "tally" in an efficient manner. In processing the next word of the designated sequence, "tally" updates "t_retrieve" and "t_howmany" as dependent on whether the word already appears in "t_retrieve".

The major design decision underlying "query" is to provide mechanisms that can determine very rapidly whether a word has a prior appearance. A hash-address scheme is utilized to decrease the average search time for a word, as compared to linear or logarithmic searching techniques.

This machine also provides the facilities for storing the designator for the current sequence and the pointer to the next word in that sequence to be processed. Let us now consider the definition of each of the modules.

The "hasher" module provides the integer parameter

hash(word w; INTEGER upper),

which for a given word w will return an integer between 0 and upper-1. It is intended that the implementation of "insert_or_increment" invoke "hash" to receive a value that is a probe into a table maintained by "query" -- see below. We decided to isolate the hashing function in its own module for essentially the same reasons that "truncator" is a separate module. Any modifications to the algorithm that implements hash(w, upper), say to more evenly scatter the probes, has no bearing on the functional behavior of the other modules of this machine. The reader will note that "hasher" appears in all machines below level4. In order to simplify the exposition, we are assuming that the primitive machine has a built-in hash function. Clearly, we could have written a program to compute a hash address, but this would contribute little to an explanation of HDM.

The "query" module provides the mechanisms for storing words and the count associated with each word, and for carrying out a hash-based search for the presence of a stored word.

The major decisions underlying the module, including its relationship with "hasher", are illustrated in Figures IV-2 and IV-3. "Query" maintains three tables corresponding to the following V-functions:

Figure IV-2: Hash Accessing of Words in "Query" -- 1

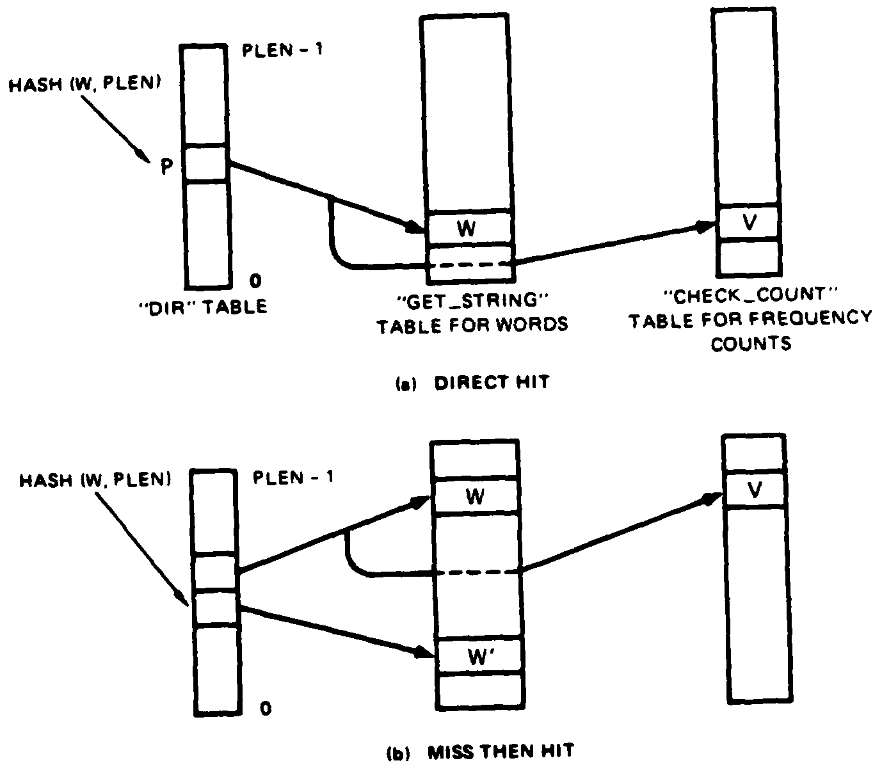
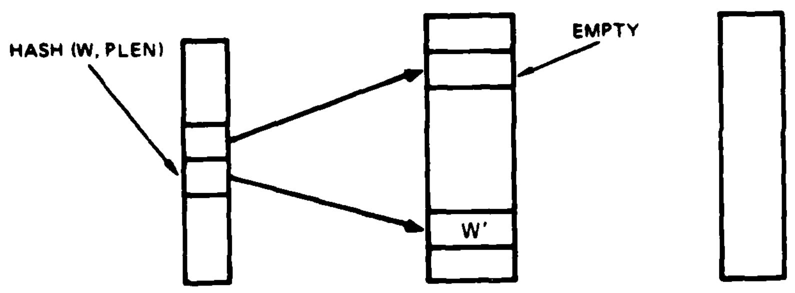


Figure IV-3: Hash Accessing of Words in "Query" -- 2



(c) MISS THEN EMPTY SLOT

get_string(INTEGER j) -> word w -- a primitive, visible V-function that is in correspondence with "t_retrieve" of "tally".

check_count(INTEGER j) -> INTEGER -- a primitive, visible V-function that is in correspondence with "t_howmany" of "tally". Remember, the count for the word get_string(i) is check_count(i).

dir(INTEGER p) -> INTEGER v -- a hash table whose entries are indices into the get_string table. The intended domain of p is between 0 and plen-1, where plen is a parameter of the module.

Each word in "get_string" is pointed to by the integer in some position p of "dir", where p is determined by a hash strategy. If a word w is somewhere in "get_string", then w can be found by probing slots in "dir" and using the value contained in the probed dir "slot" to index "get_string". The probe function for a given word w is defined as

```
probe(w, 1) = hash(w, plen)
probe(w, i) = probe(w, i-1)+1 mod plen
```

Thus, if w is in the table, then there is a j such that

```
w = get_string(dir(probe(w,j)))
```

A "direct hit" corresponds to j=1 and is illustrated in Figure IV-2a. A "miss then hit" corresponds to j=2 and is illustrated in Figure IV-2b. Finally, a "miss then empty slot" is illustrated in Figure IV-2c.

A new word w' is inserted into "get_string" only if it is not already there. Thus, there can only be one j such that

```
w' = get_string(dir(probe(w',j))).
```

When w' is inserted into "get_string", its index into "get_string" is stored in "dir" at the first empty slot starting at position hash(w,plen).

Among the decisions we have not determined so far are:

* How to represent an empty slot

* How to handle a table that is "full"

The operations supplied by this module are the following:

size() -> INTEGER v -- a visible, derived V-function that returns the number of words stored in "get_string".

save(word w; INTEGER p) -- the O-function used to store a new word at the end of the "get_string" table and put its "get_string" index into position p of "dir".

add_count(INTEGER p) -- an O-function used to increment the frequency count associated with the word in position p of "dir". Presumably, it would be invoked when it has been determined that the word of interest is already in "get_string" and is pointed to by the position p.

swap_query(INTEGER i, j) -- an O-function used to interchange the words and frequency counts stored in positions i and j of both "get_string" and "check_count".

reset_query() -- returns "query" to its initial state.

The purpose of the extremely simple "seq_pointer_cells" module is to provide storage for a sequence designator and an integer. The former corresponds to the designated sequence for which the histogram is being computed, and the latter to the pointer to the next word of that sequence to be processed. It is convenient to incorporate these facilities in a module that is separate from the other modules of the machine. For convenience, we will view the module as providing two abstract cells for storage. The access V-functions for the sequence designator and the integer pointer are respectively get_s() and get_p(); correspondingly, the respective store O-functions are store_s(seq s) and store_p(INTEGER i).

Note that the module "truncator" appears at this level, but only because we decided that it, similar to "hasher," is to be a primitive module of the system. That is, its behavior -- the value of "maxlength" -- is beyond the control of a user of the system.

C. Level3

This machine consists of the five modules "sequences", "seq_pointer_cells", "truncator", "hasher", and "intarrays". The last ("intarrays") is hidden above this level. Again, "truncator" and "hasher" are not significant to this discussion since they appear in level3 only because the are needed at a higher level and are also primitive. The module "seq_pointer_cells" also requires no discussion here since it is not used by level4, and is not implemented until level2.

The only role served by level3 is to implement "query" using a very straightforward representation. The primitive V-functions "dir" and "check_count" are each represented by an integer array of the module "intarrays". The function "get_string" is represented by a sequence of the module "sequences". This particular sequence will be in a partition of "sequences" that is only accessible to "query"; it is not used to represent a user file-system.

D. Level2

This machine contains four modules relevant to this discussion -- "vcarrays", "vc_intarray_pairs", "vc_etc_cells", and "intarrays" -- in addition to the modules "truncator" and "hasher". The purpose of the machine is to provide the mechanisms to implement "sequences" and "seq_pointer_cells". First, let us consider the definition of "vcarrays" and its role (together with "intarrays") in implementing sequences.

The "vcarrays" module maintains a collection of variable-length character-arrays, each of which is identified by a "vcarray" designator. A vcarray can expand or shrink but only from the end.

Before we describe the functions provided by "vcarrays", we discuss how sequences are implemented. Each sequence is represented by a separate vcarray and intarray. The characters for a given sequence word occupy contiguous positions in the vcarray. The start and end positions in the vcarray for the j-th word in the sequence are given by the integers in positions 2j-1 and 2j in the intarray. Essentially, the intarray is used as a directory structure for accessing words in the vcarray. Thus, the interchanging of two words in the sequence is implemented as exchanging their respective boundary pointers in the intarray.

An additional integer array is provided for the entire collection of sequences. The i-th position in this array stores the number of words in the i-th sequence (see below). The maximum number of sequences allowed is given by "v_int_bounds", a parameter of the module.

The V-functions provided by the module are the following:

char(vcarray n; INTEGER i) -> CHAR c -- returns the i-th character in vcarray n.

int_for_vcarray(vcarray n) -> INTEGER v -- returns a unique integer for each vcarray. This integer, as noted below, serves as a pointer from n to a position in the integer array for the collection of sequences. Thus, the length of sequence s, represented (in part) by vcarray n, is found in position int_for_vcarray(n) of this "collective" intarray.

The state changing operations are:

create_vcarray() -> vcarray n -- an OV-function that creates and initializes a new vcarray designator.

one_more_char(vcarray n; CHAR c) -- an O-function that adds a character to the end of the given vcarray.

remove_chars(vcarray n; INTEGER i) -- an O-function that removes i characters from the end of vcarray n.

clear_vcarray(vcarray n) -- an O-function that resets the given vcarray.

We next discuss the "vc_intarray_pairs" module. Above we indicated that a sequence is represented by a vcarray and an intarray. In terms of SPECIAL each seq designator is represented as a STRUCT with two components, a vcarray designator and an intarray designator. The type "vc_intarray_pair" has as values all such structures. Note that only a subset of all such structures are admissible as representations for seq designators; in particular, the vcarray and intarray components in any two admissible structures must be respectively distinct. This property reflects our decision to have a unique vcarray and intarray for each sequence. The module "vc_intarray_pairs" records all of the structures that have been allocated as representations for sequences. Its use in the system is primarily for proof. However, the O-function

create_vc_intarray_pair () -> vc_intarray_pair vnp

provides a morsel of procedure abstraction, by being a syntactic substitute for successive invocations of create_vcarray(); create_intarray().

The purpose of the "vc_etc_cells" module is to represent "seq_pointer_cells". Recall, "seq_pointer_cells" holds an integer and a seq designator. Since a seq designator is represented by a vcarray-intarray designator pair, we see that "vc_etc_cells" provides three cells: for an integer, a vcarray designator, and an intarray designator. (The use of "etc." in the module name is merely to keep the module name reasonably short.)

The representation of sequences in terms of these modules is described graphically in Figure IV-4.

E. Level1 Revisited

In Chapter 3 we discussed the definition of the level1 machine, formulating the decision that "chararrays" and "intarrays" are to be primitive modules. At that point in the design process one could not foresee the use of these modules in the design, nor the justification for the other modules of level1: "chararrays_intarray_pairs", "chararrays_etc_cells", "hasher", and "truncator". We have previously justified the appearance of the latter two modules. We will temporarily defer discussion of the former two in favor of indicating the role of "chararrays" and "intarrays" in implementing "vcarrays".

The representation of a vcarray is depicted in Figure IV-5. Each vcarray designator vj is represented by a unique chararray designator nj. Recall that vcarrays are of variable length while chararrays are of fixed length; thus, the length of a vcarray is bounded by the fixed length of a chararray. The length of a vcarray from level2 is represented at level1 through an integer array that is provided for the collection of chararray designators. Suppose vcarray vj is represented by chararray nj. Then, the length of vj is given by the value contained in position int_for_chararray(nj) of this "collective" integer array, where

int_for_chararray(chararray nj) -> INTEGER v

is an additional V-function to the "chararrays" module not mentioned in

Figure IV-4: Representation of a Sequence

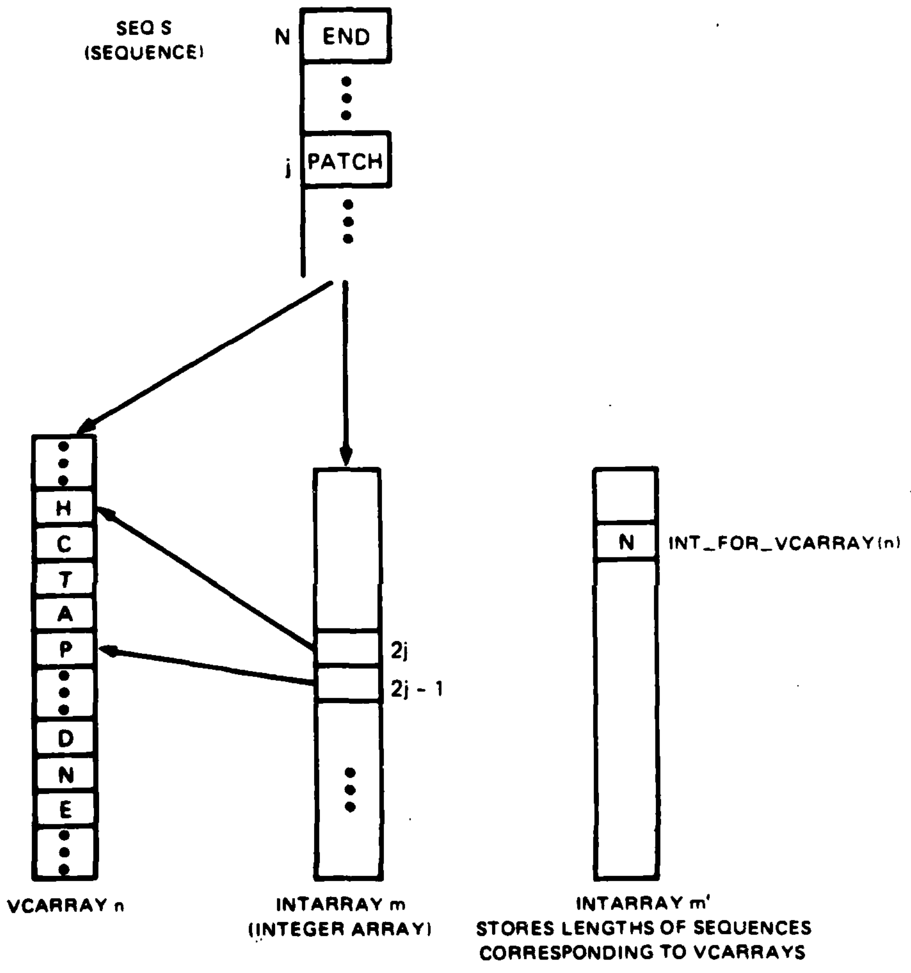
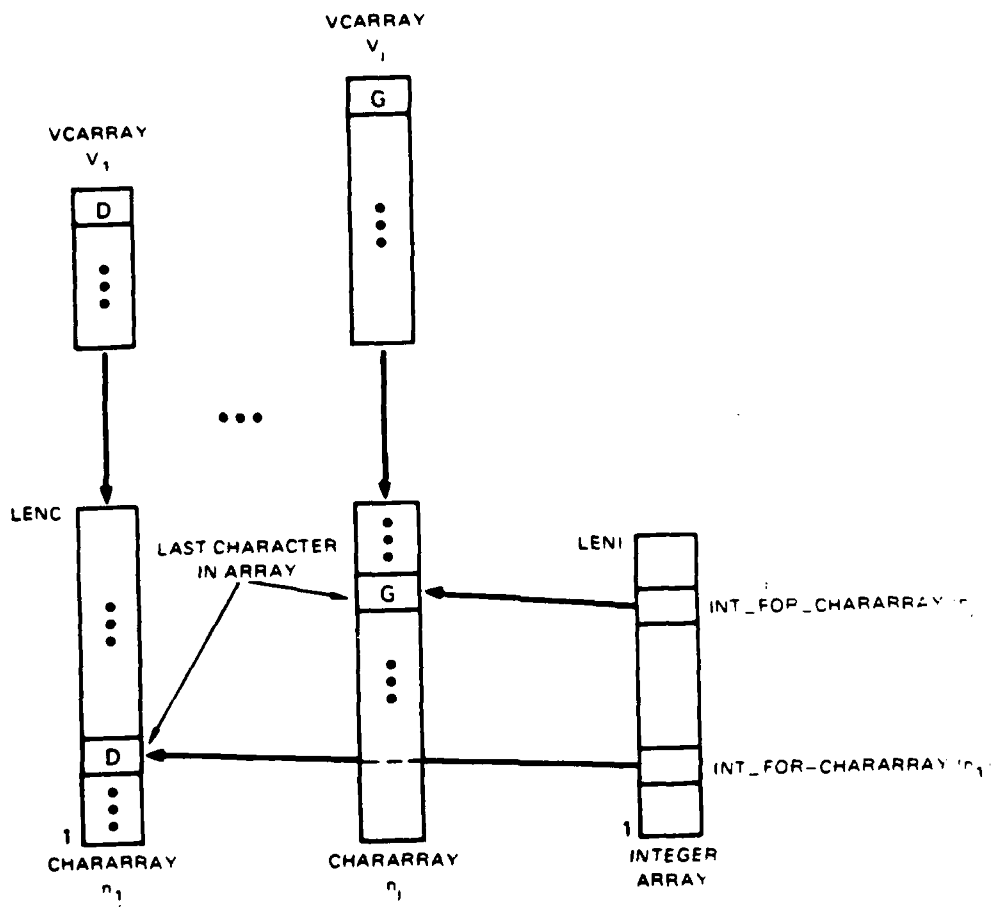


Figure IV-5: Representation of a Vcarray



our previous discussion.

The "chararrays_intarray_pairs" module stores a chararrays_intarray_pair corresponding to each vc_intarray_pair of the level2 machine. Again, this module is not needed except that the proof requires all of the pairs to be conceptually recorded.

The "chararrays_etc_cells" module is used to implement the module "vc_etc_cells". Recall, "vc_etc_cells" holds an integer, a chararray designator, and an intarray designator. Since a vcarray designator is represented at level1 by a chararray designator, we see that "chararrays_etc_cells" provides cells for an integer, a chararray designator, and an intarray designator.

In the next chapter we present the module specifications.

V SPECIFICATION OF MODULES: STAGE 4

In this stage each of the modules in the system is specified in SPECIAL. The intent of a specification is to completely characterize the functional behavior of the module. One point we want to emphasize is that SPECIAL is not a programming language. Certainly, one can produce efficient programs for a module that *satisfy its specifications*; that is the goal of the entire system development. However, we do not currently envision a compiler carrying out the translation from specifications to efficient implementation code. This is currently a task that requires significant creativity, and is likely to remain so for some time.

Below we present the specifications for each module, and in the process justify the design decisions that underlie the specifications. A specification for each machine can be derived by collecting the specifications of all modules that comprise the machine.

A. Plan for Presentation of Specifications

In discussing the module specifications we will start with the user interface and proceed downward through the hierarchy. Within a machine the order will be to proceed generally upward through the external references' partial ordering. A module specification is easier to comprehend if all other specifications on which it depends have been previously understood. For each specification, we first present in English the major decisions revealed in the specification (and not in the previous stages), and then explain the details of the specification, justifying its particular form as compared with alternatives.

The actual specifications can be found in the Appendix.

B. Sequences

Recall that this module maintains a collection of word files (sequences), each of which is identified by a unique "seq" designator. The capabilities provided by this module are as follows:

* A user of the module can request the creation of a new sequence.

* An existing sequence can be cleared to its initial state but never be deleted, i.e., there is no recycling of "seq" designators.

* The words of a sequence are read-accessed by position.

* A sequence is grown by appending words to the end.

* Two words of a sequence can be interchanged.

In this stage the major new decisions are:

* The length of a sequence is the number of non-undefined words in the sequence.

* A newly created sequence is initialized to have length zero, i.e., all words are undefined.

* Sequence words occupy consecutive positions starting at position one. Thus, if the length of a sequence is i, then the word in position i+1 is undefined; furthermore, a word is appended by placing it in position i+1.

* There is no specified upper limit on sequence word lengths.

* The maximum number of words allowed in a sequence is not specified here, but rather is governed by the available resources at lower levels.

Let us now consider how the specifications -- as given in Table 3 in the Appendix -- disclose these decisions (in addition to those informally discussed when the module's functions were introduced in Stage 1).

In the TYPES paragraph we declare "seq" as a designator type and "word" as a type whose value set contains all character vectors with positive length.

Now let us consider the function specifications in turn.

The primitive V-function

```
string(seq n; INTEGER j) -> word w
```

has single exception, which corresponds to no word being present at position j. The expression in the INITIALLY section, "w = ?" is shorthand for "initially, for all sequences the value of all positions is ?". The reader might question the absence of any exception condition corresponding to the formal argument n. What if a user invokes "string(n', j)" with some designator n' that is not an existing seq, possibly being of a different type? Is it not necessary to intercept such an exception? The following items indicate, respectively, why there is no need to check via an exception for an argument of the wrong type or for a seq that does not exist.

* Both SPECIAL and ILPL are strongly typed languages; specifications or implementations with type-mismatches are not well-formed and have no meaning. Such type-mismatches would be mechanically detected by the HDM tools, and hence would not require any handling at run-time by the exception-handling mechanism.

* In this module, a seq designator is returned only via invocation of "create_seq", and moreover, an existing sequence is never deleted. Since the protection rules for designators, embodied in ILPL, prevent the modification of a designator, any seq designator passed as an actual argument must be valid.

The visible, derived V-function

seqlen(seq n) -> INTEGER v

is expressed as the cardinality of the set that contains the indices of all non-undefined words. Several interesting aspects of this specification can be noted.

* It is emphasized that this is a specification for determining the number of words in a sequence. It is not an implementation, which can be (and is) simply carried out by using a memory cell to hold the current sequence length.

* "Seqlen" could have been defined as a primitive, visible V-function, thus avoiding the need for this "derivation" expression. However, a more complex mapping function for this module would then be required, namely to exhibit a representation for the additional primitive V-function. Generally, the number of primitive V-functions in a module should be minimized.

The purpose of the OV-function

create_seq() -> seq n

is to create a new sequence and return the newly created designator n. First, let us consider the single exception. Sometimes there is insufficient information in a module to express conditions for the occurrence of an exception. This typically occurs when the implementation of an O- or OV-function requests resources of the module(s) it invokes. In lieu of precisely specifying the conditions for an exception, we use the term "RESOURCE_ERROR" to indicate that the invocation of the function could not be completed due to some exhaustion of resources in a lower level. For an invocation of "create_seq", the RESOURCE_ERROR exception could be caused by the exhaustion of lower level (below "sequences") resources, but we choose to keep hidden from the caller of "create_seq" the exact source of the exception.

It is of interest to note that the exception could be stated entirely in terms of objects of "sequences" by providing (1) a parameter, "maxsequences", intended to indicate the maximum number of sequences that the module can support, and (2) a hidden V-function "seq_exists(seq n) -> BOOLEAN b" that, in effect, records all of the designators that correspond to known sequences. The initial value of "seq_exists" is FALSE for all n. The EFFECTS section of "create_seq" would be augmented with the effect "seq_exists(*n*) = TRUE". Thus, the exception RESOURCE_ERROR in "create_seq" would be replaced by the expression

```
            CARDINALITY({seq n | seq_exists(n) = TRUE})
                   >= maxsequences.
```

Clearly, this augmentation increases the length of the specification, albeit not significantly. However, a more serious difficulty is confronted in Stage 5 when it is necessary to map "maxsequences", say, in terms of the parameters of lower level modules: "vcarrays", "intarrays". (Let us assume that these ~~modules~~ provided, respectively, the parameters "maxvcarrays" and "maxintarrays"). One problem that occurs is that the representation of both "sequences" and "query" use integer arrays. In order to derive a representation for "maxsequences" it is necessary to pre-allocate the supply of integer arrays between "sequences" and "query". For this example, the allocation is easy: two integer arrays for "query", the rest for "sequences". However, in many systems that allow for dynamic creation of objects, the division will not be simple, and any pre-allocation is likely to result in inefficient use of the resources. Thus, we generally advocate not pre-allocating the objects of a module among modules dynamically competing for them; rather, we use RESOURCE_ERROR to trap exception conditions of such dynamic behavior. Another justification for RESOURCE_ERROR is that it defers the resolution of exception conditions to the lower levels of the system.

To express as an "effect" the generation of a never-previously-generated seq designator, we use the notation

NEW(seq).

NEW is a predefined function in SPECIAL that requires an argument of type DESIGNATOR. As part of the underlying semantics of NEW, it never returns "?".

Now consider the single expression in the EFFECTS section. It might not be immediately clear that an invocation of "create_seq" causes a state change in "sequences", and consequently, that "create_seq" should be an OV-function rather than a V-function. However, there is an underlying state involving the designators, since the designator value returned by an invocation of "create_seq" is dependent on previous invocations. One might view the module as containing a primitive hidden V-function:

```
available_sequences() -> SET_OF seq seqset,
```

which is initialized with some set of non-? designators. In lieu of the effect we have specified for "create_seq()", we could then specify:

```
n INSET available_sequences();
NOT(n INSET 'available_sequences());
```

indicating that n was in the set of available sequences designators before the invocation, but is not longer available after the invocation. Note that these effects are non-deterministic since they do not specify exactly which designator is returned. In the current version of SPECIAL we decided to use the NEW construct as syntactic sugar, since the selection of a previously unused designator is so common in specifications.

One final note about the specification of "create_seq" concerns the (apparent) absence of any effect to express the initialization of a newly created sequence. Such an expression is not needed here since the initial value of "string(n, j)" is "?", which is precisely what is desired of a sequence after it is "created". Thus, the act of creating a sequence is to make a seq designator n available so that words can be appended to n, swapped and subsequently read out.

The purpose of the O-function

clear_seq(seq n)

is to remove all words from a sequence. We express this effect by indicating that the post-invocation value in all positions of the sequence is to be "?". An equivalent, but possibly less desirable specification is

```
FORALL INTEGER j INSET {1..seqlen(n)}:
       'string(n, j) = ?
```

indicating that all positions in the sequence that previously stored "defined" words, will have value "?" after the invocation. The reader should note that in a specification conciseness is desirable, as contrasted with an implementation where efficiency is generally vital.

The purpose of the O-function

append(seq n; word w)

is to place word w at the end of sequence n. As the effect indicates, word w will be placed at position "seqlen(n) + 1", which is the post-invocation end-position of the sequence. This specification illustrates the purposeful omission in the EFFECTS section of V-function positions whose values are left unchanged. The following expressions are implicit:

```
FORALL INTEGER j ~= seqlen(n) + 1:
   'string(n, j) = string(n, j);

FORALL INTEGER j; seq n1 ~= n:
   'string(n1, j) = string(n, j)
```

The first expression indicates that all positions of n except seqlen(n) + 1 are left unchanged, and the second that all positions of all other sequences are left unchanged.

Let us now consider the possible sources of a RESOURCE_ERROR exception. Recall the decision of Stage 3 to represent a sequence in terms of (1) a vcarray to hold the characters of all words in the sequence, and (2) an integer array to hold the boundary positions of the words. Thus, the implementation of an attempt to append a new word to a sequence will require the appending of characters to some vcarray and the appending of new integers into some integer array. Either of these attempts could fail because of lack of room. It is possible to reflect both of these resource limitations in terms of (proposed) parameters of "sequences"; for the "vcarrays" limitation the "sequences" parameter would be "maxcharacters", the total number of characters allowed; for the "intarrays" limitation, the "sequences" parameter would be "maxwords". With these parameters, the specification of "append" would not require the exception RESOURCE_ERROR. However, we believe

* These augmentations needlessly clutter the specifications.
* The user of "append" probably does not need to know the cause of exception (e.g., too many words, too many total characters, etc.).
* As for "create_seq", the exception is best handled at the lowest level possible.

The specification of the O-function

swap_seq(seq n; INTEGER i, j)

should be self-explanatory. Note that no order of operation is implied in the EFFECTS section. After an invocation of "swap-seq" both expressions will be TRUE. There is no relevant "intermediate" state.

A few concluding comments on the specification for the "sequences" module can be noted. We previously mentioned that "?" is only to be viewed as a specification constant indicating "no value"; it is never to be returned as the result of a visible V-function or OV-function invocation. The semantics of NEW preclude "create_seq" from returning "?". What about for "string" and "seqlen"? It is possible to prove, based on the module specifications, that the "defined" words of a sequence n are stored contiguously in the positions 1,...,seqlen(n). This property is expressed as

```
FORALL seq n; INTEGER j:
   IF j INSET {1..seqlen(n)}
      THEN string(n,j)~=?
      ELSE string(n,j)=?;
```

These expressions are illustrations of global assertions that are properties of a module specification. Based on these global assertions, the reader should convince himself that an invocation of "string(n, j)" either raises an exception when the word in position j is "?" or returns a bona fide word. By the semantics of CARDINALITY, "seqlen" never returns "?".

We previously indicated that a designator is protected, i.e., designators may be manipulated only by functions in the module that provides such designators. No other operation can be invoked to modify a designator, and a designator cannot be typed in at a terminal. Thus, designators serve as internal system names for objects. For external access to the sequences, we envisage another module of the user interface, not discussed in this report, denoted as "name_space_manager". Via this module a user could give names to his files, and the module would maintain hidden tables that store the correspondence between usernames and sequence designators.

C. Truncator

This module specification, as given in Table 4, has four paragraphs. The TYPES paragraph declares the subtypes "word" and "truncated_word", which are used only in the FUNCTIONS paragraph.

The PARAMETERS paragraph declares the single parameter "maxlength", which (as previously indicated) is the length beyond which words are truncated for processing by "histogram".

The ASSERTIONS paragraph, in general, contains boolean-valued expressions that are constraints on the values of parameters. Such constraints must be satisfied by the initialization program that binds values to the parameters. The assertion here specifies "maxlength" to be positive, thus precluding an inconsistent specification for a truncated word -- see the "histogram" specification.

The perceptive reader might note that the desired constraint on maxlength could be incorporated in an appropriate type declaration. In lieu of the assertion, we could declare maxlength to be of type positive_integer, defined as follows in the TYPES paragraph:

positive_integer: {INTEGER j ¦ j > 0}.

The FUNCTIONS paragraph contains the specification for the derived V-function "truncation(word w) -> truncated_word tw". "Truncation" simply returns the first "maxlength" characters of an argument word; i.e., it converts a word into a truncated_word.

D. Histogram

Table 5 depicts the specifications for "histogram". Recall that the primary purpose of the module is to allow a user to select a sequence (of the "sequences" module) from which two tables are formed: "getword", which stores each distinct word of the sequence; and "howmany", which stores the number of occurrences of each word in the sequence. For processing by "histogram", only truncated words are considered. The specification at first glance appears to be complicated -- indeed, it is the most complex of the modules of the system -- but, most of complexity is due to the definitions that aid in structuring the specification and in enhancing the possibility for modification of design decisions, and to the comments that provide informal explanations.

First let us list the major decisions that are captured in the specifications. We omit those previously formulated in Stage 2.

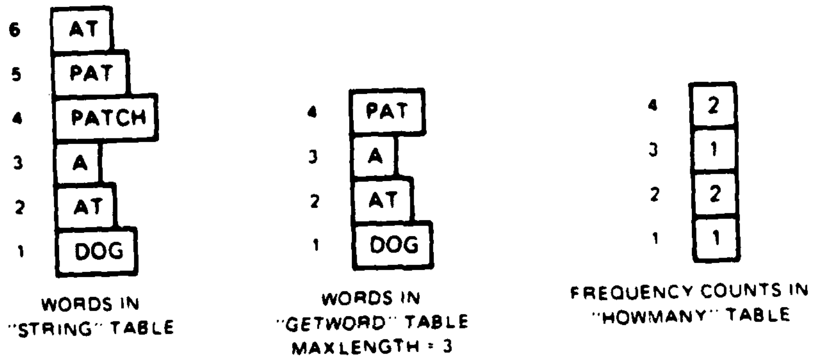
* Words are to be stored in contiguous positions of the "getword" table.

* Words in "getword" are ordered by their first appearance in the selected sequence s. More precisely, suppose that tw1 and tw2 are two distinct truncated words. If the first occurrence in s of a word w1 such that truncation(w1) = tw1 appears before the first occurrence in s of a word w2 such that truncation(w2) = tw2, then tw1 precedes tw2 in "getword".

* A histogram is to be formed for a selected sequence only if the "getword" and "howmany" tables are in their initial state. This decision is explained below.

* It should be relatively easy to change the ordering criterion for the storage of words in "getword".

Figure V-1: Histogram Tables -- Ordering by First Appearance



The second decision above is illustrated in Figure V-1, which is a refinement of the decisions reflected in Figure II-1.

A scan of the specification of Table 5 reveals the section EXTERNALREFS. Here all objects -- designator types, scalar types, V-functions, and parameters -- of other modules that are referenced in this specification are listed along with the modules in which they are originally defined, in the same format as their declaration in these modules. Recall that a module specification might not be self-contained; that is, the module might share design decisions with other modules (those referenced in the EXTERNALREFS section). The module "histogram" externally references: the module "truncator" via the parameter "maxlength"; and the module "sequences" via the designator type seq and the V-functions "string" and "seqlen". Note that the inclusion of the complete declarations for these externally referenced objects permits the syntactic and type conformance checking of the module without requiring the specifications of the other modules.

Now let us consider the two types declared in the TYPES section. The type "word" is the subtype as declared in "sequences"; "truncated_word" is a subtype of "word", constituting all words whose length does not exceed "maxlength". Note that the assertion in "truncator" on "maxlength" assures that the set of words in "truncated_word" is not vacuous. These two types are declared here to enhance the comprehensibility of the specification; they are not intrinsic to the specification as are designator types.

Let us proceed now to the specification of the functions. The reader should have little difficulty with the three V-functions and "clear_hist". For "getword" and "howmany", the exception precludes the return of "?" as the result of an invocation. The effect of "clear_hist" is to return the primitive V-functions "getword" and "howmany" to their initial state.

The specification for "hist(seq s)" is the most complicated of the system, and requires some explanation. First, consider the two exception conditions, which preclude the formation of a histogram for sequence s if (1) the module is not in its initial state, or (2) the implementation finds insufficient resources at the next lower level. It is not difficult to understand the reasons for including the second exception condition -- at this level it is impossible to predict if there will be sufficient resources at lower levels to hold the histogram tables.

Why do we require that the histogram be reset before proceeding? Recall that if an O- or OV-function invocation causes an exception to be raised, then the module state must be as it was before the invocation. One possible implementation for "hist" could, in sequentially processing the words of the sequence s, destroy the prior contents of the (representation of the) histogram tables. If a resource limitation is discovered in this processing, it would be required to restore the prior state as the RESOURCE_ERROR is raised thus, apparently, necessitating extra storage to save the contents of the tables. Of course, if the prior state is guaranteed to be the reset state, the restoration is trivial, just requiring an invocation of some resetting operation at the next lower level. Several comments are perhaps in order here:

* The need for the "hist_not_reset" exception was not discovered until the implementation for "hist" was considered.

* A planned change to HDM will allow the occurrences of state changes with exception returns. The notion of exception returns will be abandoned in favor of the more general concept of "return". With this modification to HDM there would be no need to reset the histogram tables before computing the histogram of a sequence.

* With the implementation we have selected the exception RESOURCE_ERROR is never generated. If there is sufficient space to store the sequence s, there is also sufficient space to compute and store its histogram. It is impossible to deduce this property from the information available in the module. Instead, the implementation optimizer is relied upon to discover unnecessary exceptions.

Now let us consider the EFFECTS section, which is meant to indicate that "howmany" and "getword" are modified to assure values as is illustrated in Figure V-1. First we explain the defined functions used in the EFFECTS section.

Occurrences(seq s; truncated_word tw) is a defined function that returns the frequency count of tw in sequence s. It may be understood in terms of the sequence s' that is derived from s by truncating each word w of s to a truncated_word tw. Then "occurrences(s, tw)" is understood to specify the number of times tw appears in s'. Our expression of this property in SPECIAL can be explained as follows:

> The result is the cardinality of the set (s1 INTER s2), where s1 is the set of integers between 1 and the length of sequence s, and s2 is the set of all integers i such that the truncation of the i-th word in sequence s (i.e., truncation(string(s,i)) or string(s',i)) is equal to tw.

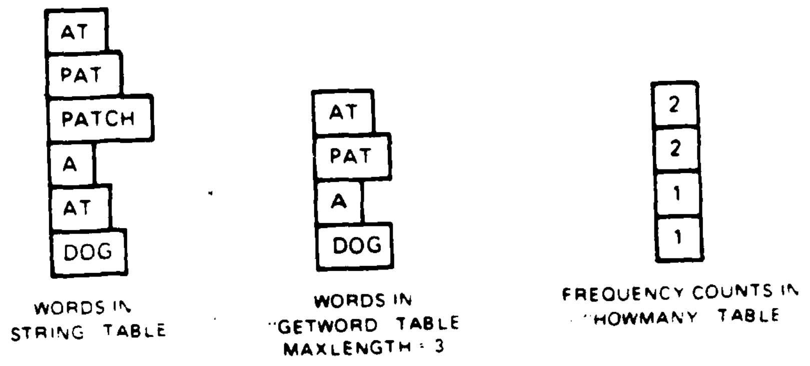
The purpose of including set s1 is to only consider "defined" words. That is we do not wish to count the number of occurrences of the word "?" in s, which is clearly unbounded.

Ith_word(seq s; INTEGER i) is a defined function that returns the i-th truncated word in sequence s' (derived from s as above) when we consider only first appearances. The definition uses the defined function "occurset(s,j)", which returns the set of all truncated words up through position j in s'. Note that occurset is defined recursively. Thus, the position in s of ith_word(s,i) is the smallest j such that the cardinality of occurset(s,j)=i. For this j, ith_word(s,i) is truncation(string(s,j)). Note that ith_word(s,k) is "?" if k is less than or equal to zero or k is greater than the number of distinct truncated words in s. The reader should convince himself he understands this.

Once an understanding of the definitions is attained, the expression in the EFFECTS section for "hist" should be clear. Note that the specified effects for "howmany" is in terms of "getword". Note also that the expression defines new values for "getword" and "howmany" for all values of j, including those for which the resultant "getword" and "howmany" values are "?". The specification is simpler than if only values of j are considered such that "getword" and "howmany" are not "?". Obviously, the implementation will restrict its consideration to the smaller subset of j.

We indicated in the "conceptualization" for this problem that, as a requirement, it should be relatively easy to modify the criterion by

Figure V-2: Histogram Tables -- Ordering by Occurrences


AT
PAT
PATCH
A
AT
DOG
WORDS IN STRING TABLE
AT
PAT
A
DOG
WORDS IN "GETWORD TABLE MAXLENGTH = 3
2
2
1
1
FREQUENCY COUNTS IN "HOWMANY TABLE

which the words are ordered in "getword". For example, suppose that the desired ordering was by the frequency count of the words in s (as depicted in Figure V-2) rather than by their first appearance. This new ordering criterion is specified merely by changing the definition of ith_word to be in terms of "occurrences".

Our specification for "histogram" could have been specified in a somewhat more elegant, although less transparent, manner using a single primitive, nidden V-function, as follows:

```
    VFUN h_howmany(truncated_word tw) -> INTEGER v;
         $(returns the number of occurrences of
              the word tw in the histogram)
         HIDDEN;
         INITIALLY
            v = 0;
```

This single V-function is intended to hold the frequency counts for all truncated_words in the universe, only a small fraction of which are actually stored in the histogram, i.e., those that have non-zero occurrence values. The EFFECTS section of "hist" becomes extremely simple, namely:

```
    FORALL truncated_word tw:
       'h_howmany(tw) = occurrences(s, tw).
```

The functions "getword" and "howmany" would become visible, derived V-functions, for which the DERIVATION sections would bear the burden of expressing the ordering of the words. The reader might wish to develop those derivations himself, using essentially the same definitions as in Table 5.

Now that we have completed our description of the modules at level6, we will proceed to level5, in particular to consider the module "tally".

E. Tally

Recall that the purpose of "tally" is to provide the mechanism for the implementation of the operations of "histogram". "Tally" implements "hist" by processing a designated sequence in a word-by-word manner. There are two primitive V-functions in "tally": "t_retrieve" and "t_howmany"; they correspond respectively to the V-functions "getword" and "howmany" in "histogram". Two other primitive V-functions in "tally", "t_sequence" and "t_pointer", respectively store the designator for the sequence s currently being processed and the position of the most recently processed word of that sequence.

The major O-function of "tally" is "insert_or_increment", which is invoked to process the next word of the sequence "t_sequence". The primary new decisions relating to the specification of "insert_or_increment" are the following:

* The truncation of the next word w, if not previously stored in "t_retrieve", is placed at the end of "t_retrieve" and its associated frequency count in "t_howmany" is set to 1. Otherwise, the count associated with the (the truncation of) w in "t_howmany" is incremented by 1.

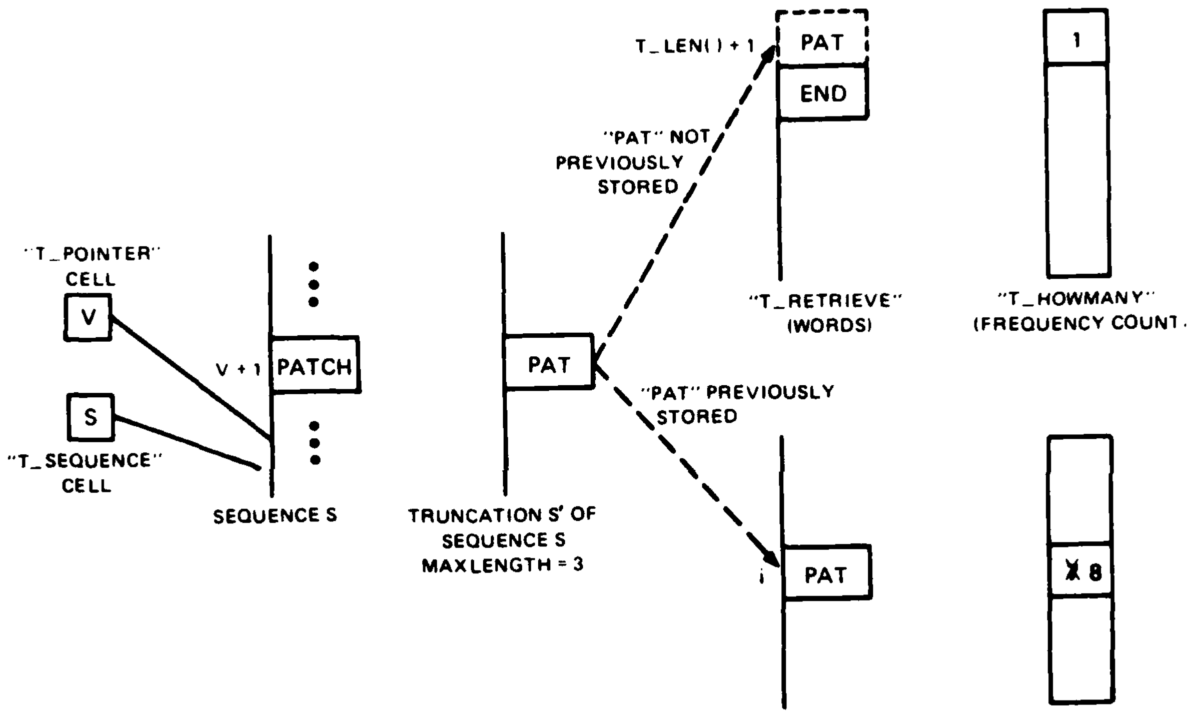
* An exception is generated if the most recently processed word was the last word of the sequence "t_sequence()", i.e., if there are no more words to process.

The major decisions underlying "tally" are illustrated in Figure V-3. The next word "PATCH" is first viewed by "insert_or_increment" as the truncated word "PAT", reflecting maxlength = 3. If "PAT" has not been previously processed, then it is specified to be placed at the end of the "t_retrieve" table and the corresponding position in "t_howmany" is set to 1. Otherwise, "PAT" has already been processed and is contained in "t_retrieve" -- let "PAT" be in t_retrieve(i); then t_howmany(i) is incremented by 1.

Let us now consider the specifications for "tally" as given in Table 6. Again, the types "word" and "truncated_word" are declared as in "histogram". The DEFINITIONS paragraph contains the definition for "no_string", which is used in a straightforward manner as an exception. The EXTERNALREFS paragraph lists the objects of "sequences" and "truncator" on which "tally" is dependent.

The V-functions' function specifications are quite straightforward. Note that the exception condition for "t_retrieve" and "t_howmany" indicates an argument that corresponds to a position that stores an "undefined" word.

Figure V-3: Decisions Underlying Tally



The O-function "t_initialize(s)" causes s to be stored in the cell "t_sequence" and 0 to be stored in the cell "t_pointer" (which represents the position of the most recently processed word).

The specifications for the O-functions "swap_tally" and "reset_tally" should be readily comprehended. As previously indicated, the formation of a histogram for a sequence s is envisioned to be a two-pass process. In the first pass, all of the distinct (truncated) words of s and their frequency counts are stored in the appropriate tables, according to the appearance of the words in s. The second pass is used to impose a different ordering, if desired. This is accomplished through repeated calls on "swap_tally".

The effects for "reset_tally" indicate that its invocation results in a transition essentially back to initial state.

Now let us consider the most interesting specification, namely that for "insert_or_increment". The specification contains an ASSERTIONS section, where assumptions on the state of the module and, perhaps, the arguments to the function are listed. The assertions listed here are unlike exception conditions, which are checked at run-time. We do not specify the behavior of the function if it is invoked with the assertions not satisfied. The one assertion to this function indicates that the sequence call "t_sequence" has been previously initialized.

The exception conditions are straightforward, corresponding respectively to no additional words in the designated sequence, and to a resource exhausting at some lower level in handling the next word of the sequence.

The first effect should pose no difficulties; it specifies that the contents of the pointer cell becomes incremented by one as a result of the invocation.

The second effect says:

> Consider some i such that position i of the "t_retrieve" table contains the truncation of the next word in the designated sequence. If such an i is found -- the word is in the table -- then the count "t_howmany(i)" associated with that word is to be incremented. On the other hand, if it has not been previously stored, as reflected by i being "?", then the next free position

of the "t_retrieve" table is to store the truncated word, and the associated frequency count is to be set to one.

It is in general possible that in a LET expression more than one value of the bound variable can satisfy the characterization. That, however, is not the case here, since as indicated by the following global assertion for "tally", a (defined) word never appears more than once in "t_retrieve":

```
FORALL truncated_word tw ~= ?:
   CARDINALITY({INTEGER i | t_retrieve(i) = tw})
   INSET {0, 1}.
```

Note that the assertion is in terms of the type "truncated_word" to emphasize that only such shortened words appear in the table.

The reader should continually convince himself that the EFFECTS are specifications, not implementations. For "insert_or_increment" the specifications indicate the changes that are to be effected to some hypothetical data structures. The specification avoids any mention of the concrete data structures, and how they are to be referenced and modified. Clearly, the specification for "insert_or_increment" hides the hash-searching strategy in terms of the structures of "hasher", "query", and "seq_pointer_cells". The latter provides the functions represent "t_pointer" and "t_sequence", and to modify them. We now discuss these three modules.

F. Hasher

The purpose of "hasher", whose specification is given in Table 7, is to provide the parameter "hash(w, upper)", which returns an integer corresponding to a word w. As indicated previously, the implementation of "insert_or_increment" will call on "hash" in searching for the location (if any) of w in the tables of "query".

The assertion indicates that for a positive value of j, hash(w, j) will be between 0 and j-1. As we will observe, the value of argument j will be "plen", the upper limit on the domain of the "query" function "dir". Note that the assertion need not exclude the word "?" as an argument since "hash" will never be invoked with "?".

It is appropriate to review the reasons for "hasher" being both a separate module and a primitive module. As we will note, the specifications of "query" and "hasher" are completely independent. By specifying "hasher" as a primitive machine we are assuming that "hash" is a primitive operation. Of course, we could have placed "hasher" at a non-primitive level and provided an implementation for "hash". In any event, the properties of "hash" specified in "hasher" that "tally" relies on are independent of the implementation. Nevertheless, the performance of the system is certainly dependent on the scattering properties of the implementation of "hash". Currently, such performance issues are beyond the scope of HDM.

G. Seq_pointer_cells

This extremely simple module, displayed in Table 8, provides storage for a seq designator and an integer, which respectively correspond to the sequence being processed and the pointer to the most recently processed word of the sequence.

The specifications should be clear, except perhaps for the motivation underlying assertion "get_s() ~= ?" for the function "get_s". [1] The value of "get_s" is initially "?", which is subsequently changed to some seq s as a result of an invocation of store_s(s), never to be "?" again. As should be clear by now, a visible function invocation is never to return "?". The assertion for "get_s" indicates that the function will never be invoked (in the implementation of "tally") when it is in its initial state. Alternatively, we could have precluded the return of "?" by providing the exception "get_s() = ?" but at the expense of providing code in the implementation of "get_s" to detect the exception in terms of lower level concepts. The "assertion" approach is generally preferred if it does not unduly constrain the use of a module.

It is perhaps appropriate to justify the need for the module

[1] Remember, the assertions section within a function specification states assumptions on the state of the module and on the values of the actual arguments when the function is invoked in an implementation.

"seq_pointer_cells". Recall that "tally" (which is at level5) provides cells for the designated sequence and its pointer. These are referenced in the specification of "insert_or_increment", which is invoked to process the next word of the designated sequence. On the other hand, "query" processes a word that is passed as an argument. The origin of the word is not important, and hence at level4 it is appropriate to separate the processing of words (and their frequency counts) from the storage of the designator of the sequence being processed and its file pointer.

H. Query

The specifications for "query" are given in Table 9. Recall that query provides two tables: "get_string" and "check_count", which correspond respectively to the "tally" tables "t_retrieve" and "t_howmany". An additional table in "query" is "dir" (for directory), which contains pointers to positions in the other two tables. The domain of "dir" is between 0 and plen-1, where plen is a parameter of "query". As illustrated in Figure V-2, if a word is stored in position j of "get_string", then j is stored in some position p of "dir", where p is determined by a hash strategy. What we have just indicated is in reality the representation of "tally", which is formulated more precisely in the next chapter. The major new design decisions for "query" introduced in the specifications are the following:

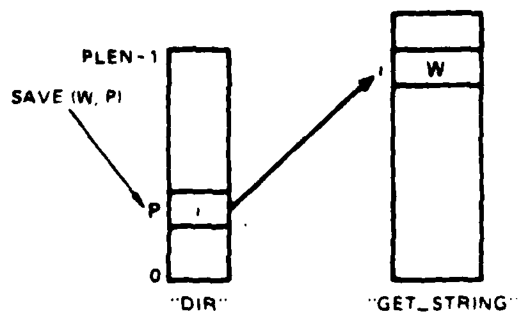
* An empty slot in the "dir" table is identified by a value of 0.

* A new word w is inserted into the "get_string" table only by invoking the function "save(w, p)". An exception is raised by "save" if (1) dir(p) already points to w in "get_string", or (2) dir(p) is non-empty and points to a word *not* equal to w. If no exceptions are raised, w is stored as the new last word of "get_string", the corresponding position is "check_count" is set to 1, and position p of the "dir" is set to point to the word position.

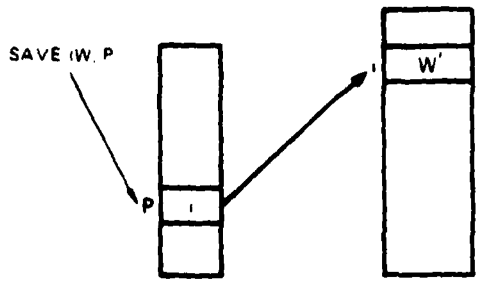
The above decisions are illustrated in Figure V-4.

Now let us consider the specifications of "query". First, note that the type truncated_word no longer appears. Although, in use

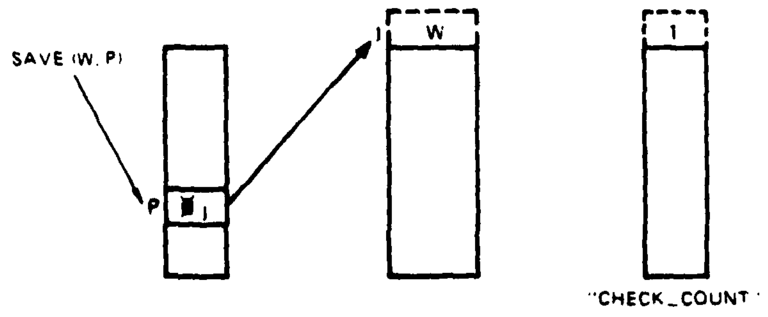
Figure V-4: Design Decisions of Query



(a) WORD IS THERE — EXCEPTION RETURNED



(b) ANOTHER WORD THERE—EXCEPTION RETURNED



(c) EMPTY SLOT — WORD STORED

"query" will only store words whose length does not exceed maxlength, it is not relevant to the specifications of "query" that the words be so constrained. Hence, these words will be considered to be just of type word.

The single assertion of the module guarantees that there is at least one slot in the "dir" table.

The reader should have no difficulty with the functions' specifications. The initial value of dir(p) is 0 only for p in the working domain of "dir" (0..plen-1); otherwise, the value is "?". For "save", two exceptions have been added to the list given above, one to make sure p is in the range 0..plen-1, and the other to trap resource errors.

The O-function add_count(p) would be invoked when it has been determined via an invocation of save(w,p) that w is already in "get_string" at location dir(p). Thus, it remains to just increment the appropriate value of "check_count", which is accomplished by invoking add_count(p). For this use of the module, the two exceptions would never be raised, and they could be replaced by the corresponding assertions:

```
p INSET {0..plen-1};
dir(p) ~= 0.
```

However, we have included the exceptions to allow a more general use of the module. Note that no RESOURCE_ERROR exception is included for "add_count", since it is expected at this point that no such exception will arise in the implementation. Actually, no exception is indicated since arbitrarily large integers are accommodated.

We have now completed all of the specifications for modules of level4. Let us now proceed to level3, which introduces the module "intarrays".

I. Intarrays

Recall that the purpose of level3 is to provide the mechanisms to implement "query". The V-function "get_string" is to be represented by a sequence, while the V-functions "dir" and "check_count" are each to be represented by an integer array. The specifications for the "intarrays" module are contained in Table 10.

A newly created integer array is initialized with a defined integer in each position. In this example all uses of integer arrays require that all positions have initial value 0. However, we have decided not to so constrain the initial value in order to illustrate a non-deterministic INITIALLY section, and to show (see Chapter VII) the format of INITIALIZATIONS programs that, in this case, zero-out the integer arrays.

Let us now proceed to level2, which introduces the modules "vcarrays", "vc_intarray_pairs", and "vc_etc_cells".

J. Vcarrays

As indicated in Figure IV-4, each sequence is represented by a vcarray and an intarray, the *former storing* all of the characters of the words, and the latter storing the endpoints of the words in the vcarray. A single "collective" integer array also stores the number of words in the various sequences.

A vcarray is a structure whose length can grow and shrink. The specifications for "vcarrays" -- given in Table 11 -- should be reasonably straightforward, except for a few subtleties. The V-function int_*for*_vcarray(n) returns a unique integer corresponding to each vcarray designator, as it is created. As indicated in Figure IV-4, this function associates a position in the sequence-length integer array with the vcarray designator that represents (in part) each seq designator. Thus, the length of the sequence represented by vcarray vc is found in position int_for_vcarray(vc) of the sequence-length intarray.

The uniqueness of int_for_vcarray(n) is embodied in the specification of

create_vcarray() -> vcarray n.

The value int_for_vcarray(n) associated with the newly created vcarray designator is specified to be in the range 1..v_int_bounds, and to be different from the value associated with any other vcarray designator. (Note that v_int_bounds is a parameter of the module.) As in the specification of "sequences", the initial values in a newly initialized vcarray remain "?".

Though the implementation of a sequence has no need for a function in "vcarrays" that returns the number of characters stored in a vcarray, the concept of the length of a vcarray is useful in the specifications of "one_more_char" and "remove_char", and hence is manifested by the definition "vclen(n)".

K. Vc_intarray_pairs

This extremely simple module -- given in Table 12 -- serves to allocate and record vcarray-intarray designator pairs, which are used to represent seq designators. As indicated in the TYPES paragraph, a "vc_intarray_pair" is a structure of the two relevant components.

The OV-function

create_vc_intarray_pair -> vc_intarray_pair vnp

is invoked to establish a new pair corresponding to a new sequence. The newly created pair is composed of a newly created vcarray designator and a newly created intarray designator. The "EFFECTS_OF" construct is used to indicate the state changes of the externally referenced modules "vcarrays" and "intarrays". Since OV-functions are invoked, the "EFFECTS_OF" statements each return a value, in addition to indicating a state change in the referenced modules.

We previously considered specifications where a specification of a module A referenced a V-function of module B. Once that V-function is declared in the EXTERNALREFS paragraph of A's specifications it can be freely referenced. If an O-or OV-function of A causes a state change in B, then there are two approaches toward specifying this state change in the EFFECTS section of that function:

* By reference to primitive V-functions of A, but quoted to reflect new values.

* By reference to O- or OV-functions of A, with appropriate

arguments. Such a reference is written as "EFFECTS_OF o(x, y)" to indicate that the new state of A is as if o(x, y) was invoked. It is not necessary that the implementation use o(x, y). Note that if multiple external references O1, O2, ..., to O- or OV-functions of A, then the new state of A is determined by considering each of the O1, O2, ..., as being applied simultaneously. The new state of A is the composition of the effects of O1, O2, ... Clearly, the composition of the effects might be inconsistent if the designer is not careful.

Often the second approach to specifying a new state for an externally referenced module leads to a simpler specification.

L. Vc_etc_cells

The purpose of "vc_etc_cells" -- displayed in Table 13 -- is to implement "seq_pointer_cells", which provides storage cells for a seq designator and an integer. Since each seq designator is to be represented by a vcarray-intarray designator pair, "vc_etc_cells" correspondingly provides single cells for a vcarray and an intarray designator. "Vc_etc_cells" also provides a corresponding integer storage cell, here called "v_get_i()".

In the case of each of the two V-functions that return designators -- "v_get()" and "v_get_n()" -- it is asserted that an invocation will only be attempted if the value of the V-function is "defined". Since there is no O-function to restore the initial state, the assertions are guaranteed to be satisfied if the corresponding "store" function is called before the V-function.

Now let us proceed to level1 for which our concern is with the newly introduced modules: "chararrays", "chararray_intarray_pairs" and "chararrays_etc_cells".

M. Chararrays

The module "chararrays" provides a fixed number of fixed-length character arrays, each of which is associated with a chararray designator. In the system, each character array holds the characters of a vcarray (variable-length character array). The cells of a single integer array, as indicated in Figure IV-5, are used to hold the current lengths of each of the vcarrays.

The major decisions embodied in the specifications of "chararrays" are:

* The length of each character array is at least one. As we will note this property permits a simple characterization in the specifications of the number of character arrays that exist at any time.

* A newly created character array will contain arbitrary characters.

* A function is included that returns a unique index into the single integer array for each chararray designator.

Let us now discuss the specifications, which are depicted in Table 14. Two integer parameters -- "maxchararrays" and "lenc" -- are provided to indicate the maximum number of character arrays that can be created and the length of a character array.

Let us skip to the specifications of the functions.

The specification of the V-function

getchar(chararray n; INTEGER j) -> CHAR c

indicates that the initial value is ?, and that an exception is to be raised if "getchar" is invoked with an argument j that is not between 1 and lenc.

The V-function

int_for_chararray(chararray n) -> INTEGER

returns a unique integer for each chararray.

An invocation of the OV-function

create_chararray() -> chararray n

is intended to return a new chararray designator n and to appropriately define a value for "int_for_chararray(n)", provided the number of previously created character arrays is less than "maxchararrays". The first effect is similar to those previously used to indicate the creation of new designator. The second effect indicates that "int_for_chararray(n)" is to return an integer between 1 and lenc, different from the integer associated with any other previously created character. The third effect specifies that all positions of the newly created character array are to be initialized with some arbitrary "defined" character. Note that the expression characterizing the exception

```
    CARDINALITY({chararray n | getchar(n, 1) ~= ?})
           >= maxchararrays
```

identifies an "existing" character array as having a "defined" character in the first position. That such a character will exist is guaranteed by the third effect, and by the assertion that the length of a character array is at least 1.

An invocation of the O-function

```
              change_char(chararray n; INTEGER j; CHAR c)
```

simply causes the j-th character of character array n to be changed to c, provided j is between 1 and lenc.

N. Chararrays_intarray_pairs

The module "vc_intarray_pairs" provides the mechanism for recording the vcarray-intarray designator pairs that correspond to existing seq designators. The vcarray is not a primitive concept, but is represented by the designator type chararray. Correspondingly, a vc_intarray_pair is represented by a chararray_intarray_pair, which consists of a chararray designator and an intarray designator. The module "chararrays_intarray_pairs" provides the mechanism needed to record all such pairs that correspond to existing vc_intarray_pairs.

The specifications -- Table 15 -- externally reference the designator types chararray and intarray. In the TYPES paragraph, the structured type chararray_intarray_pair is declared. The function specifications are as indicated below.

The V-function

```
   chararray_pair_exists(chararray_intarray_pair cnp) -> BOOLEAN b
```

returns TRUE if the pair cnp has been previously stored. The initial value of the function is FALSE for all pairs.

The O-function

```
     store_chararray_intarray_pair(chararray_intarray_pair cnp)
```

is invoked to store a particular pair, cnp. A RESOURCE_ERROR exception is provided to account for a limitation on the storage available for holding pairs.

O. Chararrays_etc_cells

This module is used to represent the module "vc_etc_cells", which holds a single vcarray designator, a single intarray designator, and a single integer value. Since a vcarray designator is to be represented by a chararray designator, "chararrays_etc_cells" provides single cells for a chararray designator, an intarray designator, and an integer.

The reader who has managed to work his way through the other thirteen modules should require no additional explanation in support of the specifications of Table 16.

This completes our discussion of the module specifications of the system. The next chapter considers the inter-machine representations.

VI MACHINE REPRESENTATIONS: STAGE 5

In this stage, we record decisions concerning the representation of the abstract data structures of each machine (except the primitive machine) in terms of those of the next lower level. In essence, one is deciding here how the data structures of each represented machine are to be assembled using the data structures of the representing machine.

It is convenient to emphasize here that, in essence, all modules of a machine are represented together. This is in contrast with the specification and implementation stages in which each module is considered separately. As we will illustrate below, there is often a sharing of decisions among the representations of the modules of machine. This sharing typically relates to the establishment of partitions of the representing machine for the modules of the represented machine.

SPECIAL is used here as the language in which representation decisions are formulated. A representation specification should be readable, concise, precise, and implementation independent. The use of non-determinism facilitates the formulation of implementation-independent representations.

In the sections below we review the format of a representation specification, and then present the overall structure of the representations for the example, the scheme for discussing the representations of each machine of the example, and a detailed discussion of each of the representations.

A. Format of Representations

This section is a condensation of the chapter on mapping functions in Volume II. Some of the paragraphs of a representation specification contain information that is redundant with that in the module specifications, but is included in the representation to permit its checking for syntactic and type consistency. Other paragraphs contain subsidiary information that aids in the structuring of the representation specification. In this section we focus on the two primary paragraphs: MAPPINGS and INVARIANTS.

1. MAPPINGS paragraph

In the MAPPINGS paragraph, the representation decisions of the upper machine are expressed in terms of the concepts of the lower machine. The information in this paragraph serves two related purposes:

1. To characterize the representation decisions for the data structures of the lower machine.
2. To permit the derivation of a mapped specification for the upper machine, i.e., a specification for each of the modules of the upper machine but in terms of concepts of the lower machine. From the mapped specification for the upper machine it is possible to derive entry and exit assertions for the purpose of verifying programs the programs that implement the operations of the upper machine. Aside from verification, a mapped specification for a module describes the functional behavior expected of the module's implementation.

Since this report is primarily concerned with the application of HDM to software development, our discussion of representations is oriented to (1) the characterization of decisions. However, it is convenient to justify the specific notation with regard to verification. Consequently, let us consider what must be contained in the MAPPINGS paragraph to permit the derivation of mapped specifications.

A machine specification is composed of expressions that reference primitive V-functions and parameters. (Sometimes the expressions are in terms of other functions, e.g., O-functions as in an "EFFECTS_OF" expression, or in terms of user-supplied definitions, but by appropriate substitution such expressions can always be written just in terms of primitive V-functions and parameters.) The MAPPINGS paragraph is to contain sufficient information to replace all references to the upper machine primitive V-functions and parameters by appropriate references to those of the lower machine. To accomplish this, we use the following format to define the mapping for each primitive V-function, V, and each parameter (which could be a parametric function) P, of the upper machine:

V(typespec$_1$ a$_1$; ...) : expr1
P(typespec$_1$ b$_1$; ...) : expr2

To the left of the colon, the upper machine function and its formal arguments are declared as in the module that defines the function. The expression on the right is in terms of lower level concepts -- primitive V-functions and parameters -- but containing references to the arguments declared on the left side. (The expression expr2 that characterizes the representation of the parameter does not reference primitive V-functions of the lower level.) What expr1 serves to characterize can be understood as follows:

* The type of expr1 is the same as that of "V", except in the case of a designator type as explained below.

* Expr1 serves as a definition for "V", with the intention that each reference to "V" in the specification for the upper machine will be replaced by expr1 in order to form the upper machine's mapped specification. From another viewpoint, expr1 indicates how each value stored in the "V" table of the upper machine is to be composed from values stored in the tables of the lower machine.

A similar interpretation applies to expr2.

As a simple example consider the primitive (hidden) V-function:

```
stack_val(INTEGER j) -> INTEGER v,
```

which returns the integer stored in the j-th location of a stack. Assume this module "stack" is to be represented in terms of a module "array" that contains the primitive V-function

```
elt(INTEGER j) -> INTEGER v.
```

The mapping for "stack_val" is to capture the following representation decision: the value of the stack pointer is to be stored in the 0-th location of the array, and the values of the stack that are "defined" are to be stored starting with location 1 of the array. Based on this decision the mapping for "stack_val" is as follows:

```
stack_val(INTEGER j) :
   IF j INSET {1 .. elt(0)}
        THEN elt(j)
        ELSE ?
```

Let us now consider the mapping of designator types. For the simple stack example, the arguments and return values of the relevant V-functions of both the upper machine and the lower machine are all of type INTEGER. Such predefined types and all constructed types ultimately composed of predefined types can be freely used in a mapping. What if the type of an argument or return value of a function of the upper machine is a designator type, or a constructed type composed of one or more designator types? Such a type has no meaning in the lower level unless the designator type is associated with a module that appears in both the upper machine and the lower machine. If the designator type does not also appear in the lower machine, we must specify a type mapping for the designator type as follows:

designator_type : type_L

where type_L is any type that can be associated with the lower machine. In essence, this mapping gives a template for the representation of each of the designators of the upper machine.

For example, assume that the stack module supported a collection of stacks, each of which is associated with a designator of type "stack", and that each such designator is to be represented by an element of the designator type "array". Then, the type mapping would simply be:

stack: array.

It is understood that each "stack" designator is to be represented by a unique "array" designator.

If the primitive V-functions for these extended modules each have an argument corresponding to the specific stack or array, then the mapping for "stack_val" would become:

```
stack_val(stack s; INTEGER j) :
    IF j INSET {1 .. elt(s, 0)}
         THEN elt(s, j)
         ELSE ?
```

Each reference to s on the right side is assumed to be for a variable of type "array_name". Note that we could have elected to represent "stack" designators via integers, as indicated by the mapping:

stack : INTEGER.

In this case, each array is identified by a unique integer, at the expense of sacrificing the protection (e.g., strong type checking) afforded by the use of designators.

Let us now consider the INVARIANTS paragraph.

2. INVARIANTS paragraph

Sometimes the use of a lower machine by the abstract programs that implement the operations of the upper machine does not involve all of the generality that the lower machine offers. This more restrictive use, not reflected by the specifications of the lower machine or the mappings, is conveniently characterized by the invariants in the INVARIANTS paragraph. Each invariant, which is a boolean-valued expression just in terms of lower-machine concepts, expresses a constraint on the values that can be acquired by V-function positions and parametric functions. (The reader might note that invariants characterize implementation decisions, and hence their formulation might be better deferred to Stage 6. At present, they are considered in Stage 5 because they can be conveniently check for consistency relative to the other paragraphs in a representation.)

As with most of the other statements of HDM, invariants serve an important role for both proof and the formulation of decisions.

* With regard to proof, each invariant can be assumed TRUE as an entry assertion to each operation of the lower level, and must be proven TRUE as a result of the operation. The use of invariants often significantly simplifies proofs of implementation.

* With regard to the characterization of decisions, invariants serve to indicate assumptions on how the lower level is to be used. Similar to the decisions of other stages, the invariants enable a dialogue between those responsible for formulating representation decisions, and those for writing the abstract programs; (both tasks could be performed by the same individual in which case the writing-down of decisions is just good bookkeeping practice). The assumptions embedded in the invariants often lead to simplifications in the programs, although it is incumbent on the programmer to ensure that the invariants are satisfied as a result of each invocation of the lower machine.

Typically there are many invariants that could be disclosed in this paragraph. However, it is recommended that only essential invariants be written-down. The essential ones are those that are the basis for simplifications in the abstract programs, or are necessary for proof. Not surprisingly, the revelation of all such essential invariants is not completed until subsequent stages are considered.

Our example system illustrates several interesting invariants.

Representations contain other paragraphs, but their detailed discussion is best discussed in the context of the example. The next section considers the overall structure of the representations for the example system.

B. Representation Structure of the Histogram Example

Figure VI-1 displays the coarse structure of the inter-module representations for the example system. We previously indicated that the representation for the upper machine MU in terms of the lower machine ML essentially considers all modules of the upper machine and the lower machine simultaneously. It is appropriate now to refine that statement. In particular, we will decompose both the represented (upper-level) and the representing (lower-level) machine into submachines for the purpose of structuring the representation.

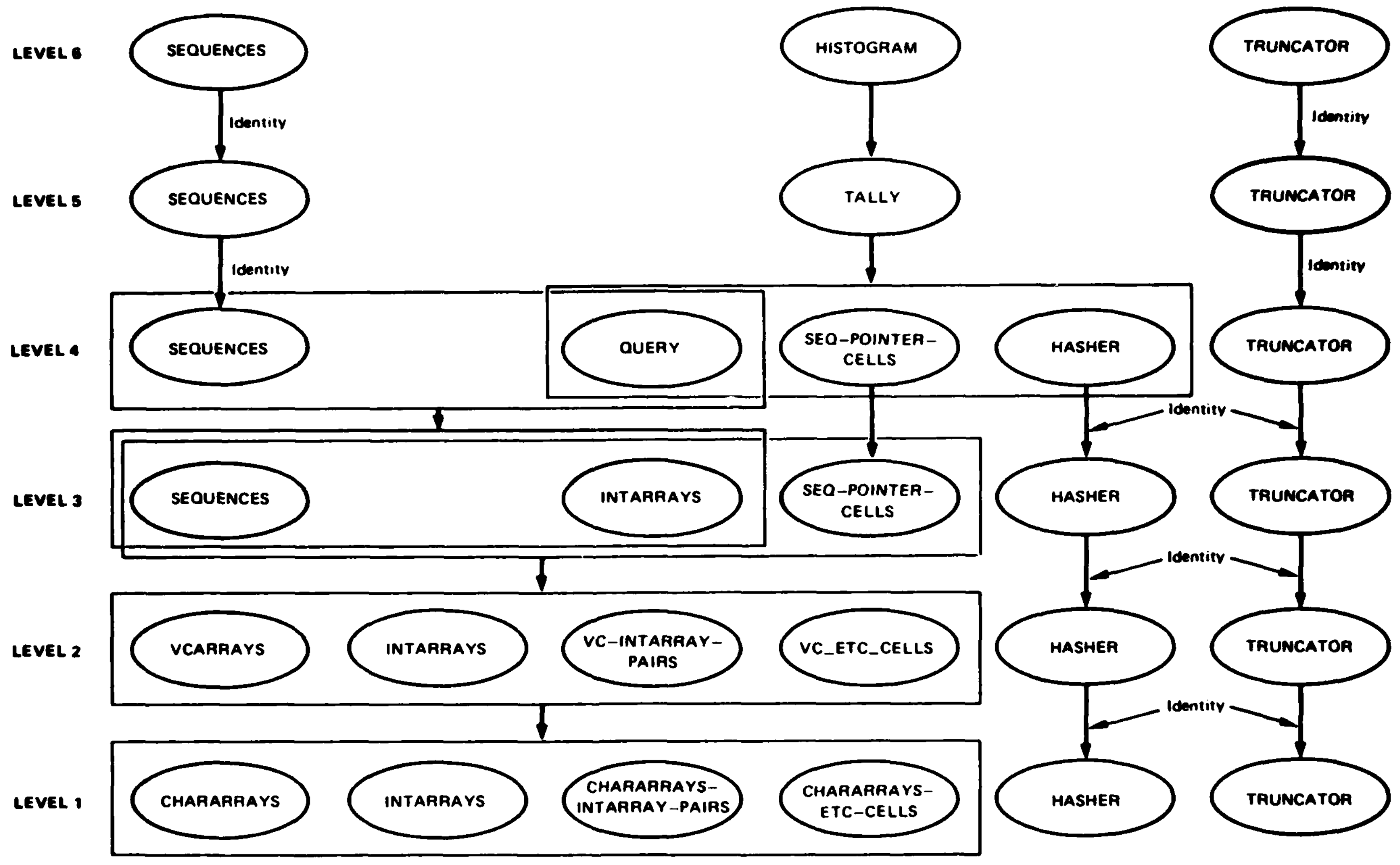
1. Structure of represented machines

First, consider the upper-level machine. In Figure VI-1, each distinct upper-level submachine is shown with an arrow emanating from it. For purposes of data representation, a given represented machine (the upper machine) can be decomposed into submachines, MU1, MU2, ... , each composed of disjoint sets of modules, such that the union of the MUi is MU. Two collections of modules, MUi, MUj, are considered as separate upper-level submachines for representation if for all pair of modules, one in MUi and one in MUj, there are not representation decisions in common. As we will note, there are several degrees to which two (upper) modules A and B can share representation decisions.

* A V-function position of some module of ML can be used to represent V-function positions of both A and B. This is the most extreme form of sharing of representation, and is usually avoided since it precludes the separate verification of the implementation of A and B. Our example does not exhibit this form of sharing.

* The V-function positions associated with a V-function of some

Figure VI-1: Coarse Structure of Machine Representations



module C in ML can be divided into two disjoint sets (partitions) -- one as part of the representation of A, and the other of B. In this case, the representations for A and B are disjoint as they bear on the implementation proofs for A and B. (In effect, C can be viewed as two distinct modules -- C' used by A and C'' used by B.) However, it is convenient to group A and B in the same submachine for purposes of representation. This form of sharing of representation decisions occurs often in applying HDM, primarily where a given module is used in the representation of two or more machines, but for simplicity is only given a single specification and implementation. For this example the designator values of "sequences" (level3) are partitioned into two subsets -- a single designator to represent "query" and the remainder to be available to a user of "sequences" above level4. To express this partitioning it is convenient to combine "sequences" and "query" into a single machine for purposes of representation.

* The same decision impacts the representation of entities in both A and B although the representations for A and B involve disjoint V-functions (possibly in different modules) of ML. For example, consider the decision to represent the type "seq" of "sequences" (level3) as a vcarray-intarray pair. This decision impacts the representation of both "sequences" and "seq_pointer_cells". For the latter module, the decision on the representation of seq designators impacts the representation for the seq-returning function "get_s". Such situations typically follow the schema: upper module A supports designator type d; upper module B contains a function that returns an object of type d; and type d does not appear at the lower level. As a result, module B must know the representation of type d, as supported by module A.

Sometimes two modules A and B are grouped together for convenience in a submachine in the absence of a shared representation decision if the declaration of the primitive V-functions of A requires a reference to module B. As we will note, this is the basis for "histogram" and "truncator" and for "tally" and "truncator" being organized as submachines, even though no representation is required for "truncator". The inclusion of "truncator" permits the syntactic checking of the representation specification without reference to other modules outside of the machine.

Figure VI-1 identifies representations between two appearances of the same module as the identity transformation. In this case, the primitive V-functions, parameters, and designator types of the upper-level appearance are mapped down identically to those in the lower-level, and no such entity is needed by any other module of the upper-level in the declaration of primitive V-functions.

2. Structure of representing machines

Now let us briefly consider the issues involved in organizing the lower-level machine into submachines for representation. Again, this decomposition enhances the understandability of the representation specifications. One can always elect not to decompose the lower-level machine.

In Figure VI-1, a representing submachine is shown as a collection of modules with an entering arrow. The basis for selecting a subset of the modules as a submachine is that at least one entity of each module is used in the representation of a submachine of the upper-level. By "use" we mean a reference in a mapping or in an invariant. It is recommended -- but not essential -- that the submachines of the lower-level be disjoint to avoid the need to demonstrate the mutual consistency of the representations that are in terms of the overlapping submachines. Note that for a machine which is both a represented machine and a representing machine (i.e., all machines except the user-interface and the primitive machine), the two decompositions need not be the same.

Before discussing the individual representations in detail, we will indicate the scheme for presenting the representations.

C. Scheme for Representation Specifications

Our plan for presenting each of the representation specifications is similar to that followed for the module specifications. First, a brief review is given of the purpose of the two submachines followed by an overview of the representation. Second, the major decisions of the representation are listed. Third, the representation specification is discussed relative to the decisions.

D. Histogram Representation

This representation is for the submachine embodying "histogram" and "truncator" in terms of the submachine containing "tally" and "truncator". Typically, the name selected for the representation is that of the most significant module of the upper-level machine. Briefly, the data structures of "histogram" are the two primitive V-functions: "get_word", which returns the "truncated_word" associated with an integer argument, and "howmany", which returns the integer (frequency count) associated with an integer argument. (Remember that "howmany(i)" is the count for "get_word(i)".) Similarly, "tally" has two corresponding primitive V-functions, "t_retrieve" and "t_howmany", in addition to two other primitive V-functions that are not involved in the representation of "histogram".

As indicated previously, a user of "histogram" has the power to create a histogram for a designated sequence by invoking a single operation. However, "tally" provides the mechanism for constructing the histogram by processing the words of the designated sequence in turn. There is no jump in data abstraction between the two modules. Instead, the difference is in procedure abstraction. Thus, there are no interesting representation decisions to discuss.

Table 17 depicts the representation as cast in SPECIAL. Since there is no data abstraction, the representation is straightforward, but does illustrate the basic paragraph structure of a representation. The first line,

```
MAP histogram TO tally
```

identifies the modules that comprise the upper-level machine, (those to the left of "TO") and the lower-level machine (those to the right of "TO").

The TYPES paragraph declares named types that are referred to in subsequent paragraphs. These types typically enhance the readability of the representation, and are often as declared in some module specification. No new designator types can be declared here. For this example, the type "truncated_word" is in terms of "maxlength" and thus necessitates the inclusion of "truncator" in both machines.

The EXTERNALREFS paragraph lists primitive V-functions, parameters, designator types, and scalar types of the modules that are "involved" in the representation. All such entities of the modules in the upper-level machine must be included since, obviously, they must be given representations. Only those entities of the lower-level machine that are referred to in the representation need be included. The information given for each primitive V-function is taken from the header of the module specification that includes this function. Similar information is provided for the parameters and designator types.

The basic format of the MAPPINGS paragraph was discussed in a previous section. A mapping must be provided for each entity of the upper-level modules. In this case the mappings are trivial, reflecting the absence of data abstraction. Our requirement that each upper-level entity in the EXTERNALREFS paragraph be mapped is the reason for included the trivial mapping for "maxlength".

Now let us proceed to the representation for level5.

E. Tally Representation

This representation is for the represented cluster consisting of "tally" (level5) in terms of the level4 representing cluster consisting of "query", "seq_pointer_cells", and "hasher". The primary purpose here is to represent the four primitive V-functions of "tally": "t_retrieve", "t_howmany", "t_sequence", and "t_pointer". Recall that "t_retrieve" returns a truncated_word corresponding to some integer; "t_howmany" returns an integer value (frequency count) corresponding to an integer; "t_sequence" returns a seq designator corresponding to the sequence for which the histogram is being constructed; and "t_pointer" returns an integer that identifies the location in the designated sequence of the next word to be processed.

The module "seq_pointer_cells" provides two primitive V-functions that are used to trivially represent "t_sequence" and "t_pointer". The module "query", similar to "tally", provides the mechanism to store words and their associated frequency counts. In addition, it enables the efficient determination of the existence of a word in a "query" table. A hash searching scheme is utilized here, where the module "hasher" provides a function that for a given word returns an integer in a particular range.

The major representation decisions embodied here are listed below. The first relates to the data structure representation for "tally"; the latter two to the use of level4 by the "tally" implementation. These are in reality implementation decisions, but are conveniently formulated here as invariants.

* The data structure mapping for "tally" is trivial. Each of the four primitive V-functions of "tally" are associated with a primitive V-function of either "query" or "seq_pointer_cells".

* A given word appears no more than once in the "get_string" table of "query".

* If a word w is in the j-th location of "get_string" table and if the initial hash probe corresponding to w is p, then some location in the "dir" table of "query" between p and the first empty slot contains the value j. This decision reflects the usage of the "query" data structures to accomplish hash searching.

Now let us consider the representation specification as displayed in Table 18. The EXTERNALREFS paragraph lists the relevant entities of six modules. Four of these modules are directly involved in the representation, while "sequences" and "truncator" provide entities that are just referenced. (Since "sequences" also appears in the lower level machine, its representation does not need to be known here. This situation is contrasted with our shared representation schema in the previous section.) Included are the primitive V-functions and parameters of the represented cluster ("tally") and the representing cluster ("query", "seq_pointer_cells", "hasher"), while only those entities of "sequences" and "truncator" that are actually referenced are included. For "sequences", the referenced entity is the designator type seq, used to identify the type returned by "t_sequence" and "set_s"; for "truncator", it is the parameter maxlength, used to define the type truncated_word. Again, it is emphasized that seq and maxlength are not represented nor are the targets of any representation; they are included here so that the tally representation will be self-contained for checking purposes.

The MAPPINGS paragraph indicates the trivial representation decisions for the four primitive V-functions of "tally".

The interesting aspects of the representation specification are in the INVARIANTS paragraph. Each of the invariants expresses a constraint on the state of "query" that is to be satisfied for the initial state, and after each O-function invocation. These constraints are stronger than those implied by the specifications, and reflect the non-arbitrary manner in which the O-functions of "query" are invoked. As previously indicated, the invariants express implementation decisions, but are conveniently considered at this stage since they are cast as boolean expressions in terms of the primitive V-functions, parameters, and types of the representing machine.

The first invariant:

```
FORALL word w ~= ?:
   CARDINALITY({INTEGER j | get_string (j) = w})
      <= 1,
```

indicates that no defined word is to appear more than once in the "get_string" table. By referring to the specifications for "query" -- Table 9 -- it is clear that successive invocations of "save"

```
save(w1, 0);
save(w1, 1);
```

in a newly initialized "query" module will cause the word w1 to appear in locations 0 and 1 of "get_string". By writing the above invariant we are indicating that a program implementing "tally" will never generate such a sequence of invocations of "save". Before invoking save(w1, p1) for some empty slot p1, it will be assured that w1 does not already appear in the "get_string" table.

The second invariant characterizes the hashing scheme that is employed here, and is a statement of what is depicted in Figures IV-2 and IV-3. Here, use is made of the DEFINITIONS facility to structure the invariant. First consider the defined function "probe_succeeds", which expresses the hashing condition for a word w.

```
    probe_succeeds(word w) IS
       EXISTS INTEGER p INSET {0 .. plen-1}:
           get_string( dir(p)) = w
           AND (FORALL INTEGER i INSET
                 {hash(w, plen) ..
                    IF p < hash(w, plen)
                       THEN p + plen
                       ELSE p}:
                dir(i MOD plen) ~= 0).
```

This function returns TRUE if: the word w is in location dir(p) of the "*get_string*" table for some p such that for all locations d of the "dir" table between hash(w, plen) and p (allowing for "wrap-around" if p < hash(w, plen)), it is the case that dir(d) ~= 0. That is, there are no empty slots between hash(w, plen) and p.

The second invariant

```
    FORALL word w |
       w ~= ? AND (EXISTS INTEGER j | get_string(j) = w:
                   probe_succeeds(w)),
```

expresses the restriction that all words w that appear in the "get_string" table satisfy the hashing condition.

This completes our description of the representation of "tally". The primitive V-functions of "tally" map down identically to lower level entities, so the most interesting aspect of the representation specification is in the invariants that capture the use of "query" to represent a hash searching scheme. Let us now proceed to the representation of "query".

F. Query Representation

This representation is for the represented cluster of "sequences" and "query" in terms of the representing cluster of "sequences" and "intarrays". The primary purpose here is to represent the three primitive V-functions of "query" ("dir", "get_string", and "check_count") and its parameter ("plen") in terms of the entities of "sequences" and "intarrays". As we will observe, the selected representation is quite simple; "get_string" is represented as a sequence, while the other two primitive V-functions of "query" are represented as integer arrays.

The reader might question the need to include "sequences" in the represented cluster. There is a sharing of representation decisions between "query" and the instance of "sequences" at the upper level. As we will note, the shared decision is that the particular sequence used to represent the "get_string" table is not available to the instance of "sequences" at the upper level. That is, the representation induces a partitioning of the lower level "sequences" module.

Now let us discuss the representation as displayed in Table 19. First note the PARAMETERS paragraph, which has not been confronted in the previous representations. In general, this paragraph declares constants or functions that are needed in the INVARIANTS or MAPPINGS paragraph. It is emphasized that:

* A representation parameter is only dependent on the representing modules. That is, the type of the parameter is derived as some combination of predefined types and designator types of the representing machine.

* A representation parameter remains constant. In the next chapter, we will show that representation parameters are bound to values when the upper level modules are initialized.

For this representation, three parameters are declared:

unique_string: a seq designator for the sequence that represents the "get_string" table.

director_array: an intarray designator for the integer array that represents the "dir" table.

count_array: an intarray designator for the integer array that represents the "check_count" table.

The TYPES paragraph introduces the type "seq1",

seq1: {seq n ¦ n ~= unique_string},

which is seen to be a subtype of seq of the lower level "sequences" module. That is, the type "seq1" has as values all designators of the lower level "sequences" module except the particular designator "unique_string".

Now consider the first mapping in the MAPPINGS paragraph:

"seq: seq1". The values of the (upper-level) designator type seq are represented by values of the (lower-level) designator type seq1. These values are all the seq designators of lower level sequences module except "unique_string".

The remaining mappings should be easily understood. For example, the second mapping:

```
    string(seq n; INTEGER j): string(n, j)
```

indicates that for each seq designator n of the upper level "sequences" module and for each integer j, the values of string(n, j) map down to string(n, j), where the latter n is a reference to a seq designator of the lower level "sequences" module. Thus, "string" is identically mapped.

The third mapping: "plen: leni" indicates that the "query" parameter "plen" is to be represented as the "intarrays" parameter "leni" -- the length of the fixed-length integer arrays.

The fourth mapping:

```
    get_string(INTEGER j):  string(unique_string, j)
```

indicates that the j-th word in the "get_string" table is to be represented as the j-th word in the sequence "unique_string".

The fifth mapping:

```
    dir(INTEGER p):  getint(director_array, p + 1)
```

indicates that the p-th integer in the "dir" table is to be represented as the (p + 1)-th integer in the integer array "director_array". Recall that the domain of interest for "dir" is 0..plen-1, while the domain of interest for "get_int" is 1..leni.

A single invariant is included,

```
    FORALL INTEGER j INSET {1..leni}:
      getint(director_array, j) INSET
        {0..seqlen(unique_string)},
```

which indicates that the "director_array" will only contain values between 0 and seqlen(unique_string), the number of words in the particular sequence "unique_string". This invariant reflects the use of the integer array "director_array" as a repository for either 0's or pointers to words in the sequence "unique_string".

Now let us proceed to the representation of "sequences" and other modules of level3.

G. Sequences Representation

Here we consider the representation for the level3 cluster of "sequences", "intarrays", and "seq_pointer_cells" in terms of the level2 cluster of "vcarrays", "intarrays", "vc_intarray_pairs", and "vc_etc_cells" (see Table 20). The main decision expressed here is to represent each sequence in terms of a variable length character array, i.e., in terms of a vcarray and an integer array. More explicitly, the decisions are as follows:

1. The vcarray corresponding to a sequence n will hold the characters of all words of the sequence, with no separators between words. The corresponding integer array will hold the position of characters in the vcarray that are the first and last character of each word. Figure IV-4 shows the basic representation scheme.

2. Each seq designator is to be represented as a pair consisting of a vcarray designator and an intarray designator.

3. A particular integer array, "nstrings", is set aside to hold the number of words in each sequence.

4. Assume that a sequence n is represented by a vcarray vc and integer array m. Then the position in "nstrings" that holds the number of words in sequence n is given by

   int_for_vcarray(vc),

   where "int_for_vcarray" is a primitive V-function of the "vcarrays" module that maps a vcarray designator vc to an integer. This V-function is provided by "vcarrays" in lieu of a built-in pointer facility in the HDM languages.

5. Recall that "seq_pointer_cells" provides two primitive V-functions: "get_s" and "get_p"; the former is conveniently viewed as a cell holding a seq designator. In view of decision (2), "get_s" is naturally represented in terms of two cells of "vc_etc_cells", which respectively hold a vcarray designator and an intarray designator.

6. Each of the pairs that represent seq designators are held via the V-function "vc_pair_exists" of the module "vc_intarray_pairs".

Now let us consider the representation as depicted in Table 20.

First consider the second and third named types declared in the TYPES paragraph. (The fourth declares "word" as in previously discussed module and representation specifications, while the discussion of the first named type must await elaboration of entities that comprise its definition.)

"Vc_intarray_pair" is a structure type defined as:

STRUCT_OF(vcarray vcarray_part; intarray intarray_part).

For any particular value of the type, the first component is a vcarray designator with selector "vcarray_part", while the second component is a intarray designator with selector "intarray_part". This type has as values all pairs such that the first component is a vcarray and the second is an intarray. Only a subset of these pairs are used as representations for seq designators. For example, assume the representations of seq designators n1, n2 are the respective structures (vc1, m1) and (vc2, m2), where vci is a vcarray designator, and mi is an intarray designator. Then, the structure (vc1, m2) is certainly in the type "vc_intarray_pair", but is not a representation of a seq. It is useful in the representation specification to identify, as a type, all elements of "vc_intarray_pairs" that also represent a seq.

The V-function

vc_pair_exists(vc_intarray_pair vcnp) -> BOOLEAN b

of the module "vc_intarray_pairs" is intended to keep track of all such pairs.

Thus the third named type, "vc_intarray_pair1",

{vc_intarray_pair vcnp | vc_pair_exists(vcnp) = TRUE}

has as values all pairs that, at any instant, are allocated to the representation of seq designators. Note that the values of "vc_intarray_pair1" vary dynamically. Each new creation of a sequence, which precipitates the establishment of a new representing pair, adds a new value to the type "vc_intarray_pair1".

In the PARAMETERS paragraph the intarray designator "nstrings" is declared. The integer array corresponding to "nstrings" is used to hold the number of words in each sequence. It is reasonable to declare "nstrings" as a representation parameter since it will be established when the "sequences" module is initialized. We know that there will be additional integer arrays set aside for the representation of sequences, namely one integer array for each sequence to hold the boundary positions of the representation of words. Let us denote the set of such integer array designators as "inclength", which is defined in the DEFINITIONS paragraph as follows:

```
    {intarray m | EXISTS vc_intarray_pair1 vnp :
              vnp.intarray_part = m}.
```

Thus "inclength" is the set of all intarray designators used to represent sequences.

Now we are prepared to indicate the partitioning of intarray designators of the lower level appearance of "intarrays" between "sequences" and the upper level appearance of "intarrays". The named type "intarray1", given by

```
          {intarray m | NOT m INSET(inclength UNION {nstrings})},
```

contains as values all intarray designators except those in the set "inclength" UNION {nstrings}. Thus, the first mapping, "intarray: intarray1", indicates that the intarray designators available to the upper level appearance of "intarrays" are that subset of the lower level intarray designators contained in the type intarray1. The remainder of the lower level designators are allocated to "sequences".

The second and third mappings merely indicate that the parameter "leni", and the V-function "getint" are mapped identically.

The fourth mapping, "seq: vc_intarray_pair1", expresses the decision that each seq designator is to be represented as a vcarray-intarray pair, where the intarray designator is restricted to be in the set inclength.

The fifth mapping indicates how the words of a sequence are to be formed from a vcarray and an intarray. Two defined functions are useful in formulating the mapping. First, consider

```
                    len_seq(vc_intarray_pair1 vnp),
```

given by

getint(nstrings, int_for_vcarray(vnp.vcarray_part)).

It returns the integer that corresponds to the number of words in the sequence represented by the vcarray-intarray designator pair vnp. (Recall that

int_for_vcarray(vc)

returns a unique integer corresponding to the vcarray designator vc. Uniqueness is guaranteed by the "vcarrays" module specification in Table 11.)

The second defined function is

ival(vc_intarray_pair1 vnp; INTEGER i),

given by

getint(vnp.intarray_part, i).

It returns the i-th integer in the intarray that is part of a vcarray-intarray representation of a sequence.

Now consider the mapping for the "sequences" primitive V-function "string(seq n; INTEGER j)":

```
IF j INSET {1..len_seq(n)}
   THEN VECTOR(FOR i FROM 0 TO ival(n, 2*j) - ival(n, 2*j-1):
        char(n.vcarray_part, i + ival(n, 2*j-1)))
   ELSE ?
```

The j-th word of sequence n is represented by the vector formed from consecutive characters in the representing vcarray. The boundary positions of this string of characters are found in the 2j-th and 2j+1-th positions of the representing integer array. If j is not a valid location for a word, then the word (and, of course, its representation) is "?".

The last two mappings for the primitive V-functions of "seq_pointer_cells" are trivial. For "get_p", which returns an integer, the representation is simply "v_get_i". The mapping for "get_s" is:

get_s (): STRUCT(v_get(), v_get_n()).

The representation of a seq designator (returned by "get_s") has two components: the "vcarray_part" is held by "v_get" and the "intarray_part" by "v_get_n".

This completes our description of the representation of "sequences" and related modules. Now let us proceed to the final mapping, for "vcarrays" and related modules.

H. Vcarrays Representation

In this section, we present the representation of the level2 cluster of "vcarrays", "intarrays", "vc_intarray_pairs", and "vc_etc_cells" in terms of the level1 cluster of "chararrays", "intarrays", "chararrays_intarray_pairs", and "chararrays_etc_cells". The "heart" of the representation is concerned with how each vcarray is composed from a (fixed-length) character array and a value in a particular location of an integer array. Three other modules that share representation decisions with "vcarrays" are included in the representing cluster: "intarrays" and "vcarrays" divide up the integer arrays of the lower level appearance of "intarrays", and "vc_etc_cells" and "vc_intarray_pairs" depend on the representation of the "vcarray" designator type.

The major representation decisions expressed here are:

* Each vcarray designator is to be represented by a unique chararray designator.

* The characters in each variable-length character array are held in corresponding locations of a (fixed length) character array. A particular integer array, named "length_array", holds the locations in the character arrays of the last character of each represented vcarray. A unique position in "length_array" is associated with each character array, namely the position defined by the "chararrays" V-function "int_for_chararray(chararray n)". Figure IV-5 depicts this representation decision.

* The vcarray-returning function "v_get()" of "vc_etc_cells" is represented by the chararray-returning function "c_get()" of "chararrays".

* A vcarray-intarray designator pair (i.e., a vc_intarray_pair) is represented by a pair consisting, respectively, of the corresponding chararray designator and the same intarray designator.

Now we are ready to discuss the details of the representation specification appearing in Table 21. It is convenient to discuss the represented modules in the following order: "intarrays", "vcarrays", "vc_etc_cells", and "vc_intarray_pairs".

A single integer array with designator "length_array" is allocated to the representation of "vcarrays"; the remaining integer arrays are available to the upper level appearance of "intarrays". We first declare a subtype

intarray1: {intarray m ¦ m ~= length_array}

and then define the mapping of (the upper level appearance of) intarray to be "intarray: intarray1". The remaining entities of "intarrays", the parameter "leni" and the V-function "getint", are mapped identically.

Now consider the representation of "vcarrays". The designator type "vcarray" is simply represented by the designator type "chararray". The most interesting representation expressed here is for the "vcarrays" V-function "char", as *follows:*

```
IF j INSET {1..getint(length_array, int_for_chararray(n))}
   THEN getchar(n, j)
   ELSE ?
```

The j-th character of vcarray n is represented as getchar(n, j) provided j is in bounds (namely between 1 and the current length of the vcarray). This "length-value" is represented as the value in position int_for_chararray(n) -- n being the chararray designator that is the representation of vcarray n -- of the integer array "length_array". If j is not in bounds, then the value returned by char(n,j) is ?.

The module "vcarrays" contains the parameter "v_int_bounds", the value of which is the upper limit of the range of "int_for_vcarray(n)". Since the range of "int_for_chararray(n)" is constrained by the length of the integer arrays, it is natural to represent "v_int_bounds" as "leni".

In order to understand the need for the single invariant "leni >= maxchararrays" it is necessary to look ahead to the implementation of the "vcarrays" OV-function "create_vcarray()". As indicated in Table 11, the invocation can return a RESOURCE_ERROR. What resources of level1 modules could become exhausted?

1. Clearly, all of the available character array -- initially size "maxchararrays" -- could be used up.

2. In addition, "length_array", which holds the current lengths of each vcarray, could be exhausted.

In general, both of these conditions would have to be considered by the implementation of "create_vcarray()". However, if it is guaranteed that the availability of an additional character array implies the existence of an available slot in "length_array", the condition (2) need not be handled in the implementation.

Now let us consider the representation of "vc_etc_cells". The primitive V-function "v_get_n" returns an intarray designator; it is trivially represented by the "chararrays_etc_cells" function "c_get_n". Similarly, "v_get_i" (which returns an integer) is trivially represented by "c_get_i" (which also returns an integer). Since vcarray designators are being represented by chararray designators, it follows that "v_get" (which returns a vcarray designator) is represented by "c_get" (which returns the corresponding chararray designator).

Finally, we consider the representation of the "vc_intarray_pairs" V-function

vc_pair_exists(vc_intarray_pair vnp) -> BOOLEAN b.

Since the type "vcarray" is represented by the type "chararray", the structure type "vc_intarray_pair" is represented by a structure type whose first component is of type "chararray". This correspondence is implicit in the mapping of the designator types. We have called this lower-level structure type "chararrays_intarray_pairs". Then "vc_pair_exists" is trivially represented by the function "chararray_pair_exists".

I. Summary of Representations

Five representation specifications have been described for this example, excluding the identity mapping that is the case for several of the module representations. Most of the representations are quite simple, reflecting a small jump in data abstraction between adjacent machines. The three most interesting mappings were for:

* "Tally", illustrating the use of invariants to characterize a hash storage scheme,

* "Sequences", illustrating the representation of a word sequence in terms of a variable length character array, an integer array that indicates boundary positions for words, and a position in an integer array that holds the number of words in the sequence.

* "Vcarrays", illustrating the representation of a variable length entity in terms of those whose length is fixed.

The remaining mappings are almost trivial, either involving no real data abstraction, or involving the use of an array to represent a function.

Now we can proceed to the next stage -- implementation.

VII MODULE IMPLEMENTATION: STAGE 6

A. Introduction

In this chapter we discuss Stage 6, which is concerned with the formulation of implementation decisions. In HDM, each module in a system is to be implemented separately. For a module that appears at multiple adjacent levels, only the lowest level appearance is actually implemented. However, for convenience at this stage, each module in the system is viewed as having an implementation, although for a module m in machine M_i that also appears in M_{i-1}, the implementation will be the identity. These identity implementations are not present in the ultimately generated code -- stage 7.

In order to understand what must be expressed in a module implementation, consider what it means to invoke a visible operation of a module m of the top level machine, with actual values for arguments. Such an invocation precipitates a sequence of invocations to visible operations (or parameters) of modules that are at the next lower level or at the same level. Each such invoked operation itself precipitates a sequence of invoked operations, and so on until the operations of the primitive machine are invoked and evaluated. The processing of a top-level invocation is thus similar to the processing of a nesting of non-recursive subroutine invocations.

Thus, an implementation of a module consists of an abstract program for each operation and parameter. Each such abstract program is a shorthand description of a sequence of invocations of other operations for each invocation of the program itself. An additional program denoted as the "initializations" program serves to drive a module into its initial state. This program, understood to be executed before any operations of the module are invoked, invokes the other operations of any other module -- already initialized -- just as any other program. After the execution of the "initializations" program for a given module, the value of its primitive V-functions and parameters (as derived by applying the representation functions to the values of the primitive V-functions and parameters of the next lower level modules) will have their initial values. It is assumed that the modules of the primitive machine are in their initial state when "powered-up". Thereafter, modules are initialized in an obvious order starting with the modules at level2.

In the remainder of this section we discuss: (1) the structure of the implementations for the histogram example, (2) the scheme for presenting the implementations, and (3) a detailed discussion of each of the module implementations.

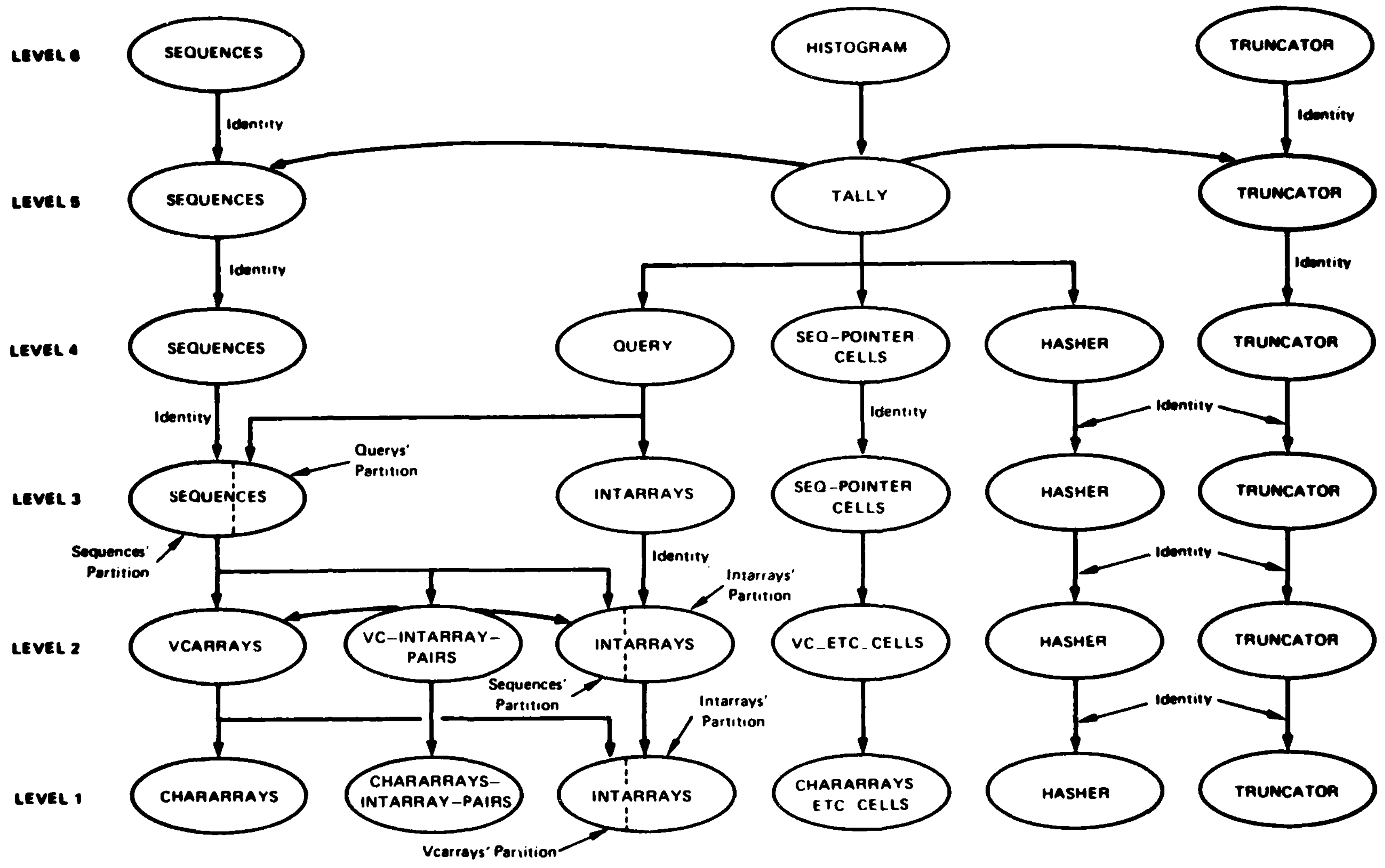
Before proceeding, the reader should review Chapter 7 (ILPL) of Volume II.

B. Structure of Implementations for Example

In order to assist the reader in following the subsequent sections, we display the coarse structure of the system implementation in Figure VII-1. In this view the abstract machines are organized as modules as in previous depictions of the system -- Figures IV-1 and VI-1 -- the latter two for the purposes of describing specifications and representations. In this view, a module sometimes appears in several machines with the "identity" implementation serving to implement an appearance in terms of the next lower level appearance. The other modules have non-trivial implementations, possibly in terms of modules at the same level -- as for "tally" and "vc_intarray_pairs" -- and several lower level modules. An arrow from module A to B indicates that in the implementation of A there is a reference to an operation or parameter of B. As we will note, the abstract programs for a module can reference designator types of a module, but not necessarily operations of that module. Such references are not explicitly depicted in the illustration.

Figure VII-1 also illustrates the partitioning of the level3 appearance of "sequences" and the level2 and level1 appearances of "intarrays". In the case of "sequences" its designator set -- and the V-function position associated with designators -- is divided into two partitions. One partition contains the seq designator "unique_string"

Figure VII-1: Structure of Implementation -- Abstract Machine View



AD-A091 272 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK. VOLUME--ETC(U)
JUN 79 W L SUTTON, K N LEVITT, L ROBINSON N00123-76-C-0195
UNCLASSIFIED NOSC-TD-366-VOL-3 NL

2 OF 2
AD
A091272

END
DATE
FILMED
12-80
DTIC

for the private use of "query2 in its implementation of the histogram; the other partition contains the remaining seq designators, all of which are ultimately available at the user interface, and correspond to word sequences that a user will directly manipulate.

Figure VII-2 depicts the implementation structure, but in a form where a module appears only once. The implementation dependencies are also clearer here, since all modules that serve to implement a module m are shown below m.

C. Scheme for Presenting Implementations

In the following sections we present the implementations for each module of the example. The discussion for each module consists of three parts as follows:

1. A brief review of the module being implemented and of the implementing modules.
2. A listing of the decisions underlying the implementation.
3. A detailed discussion of the abstract programs.

D. Histogram Implementation

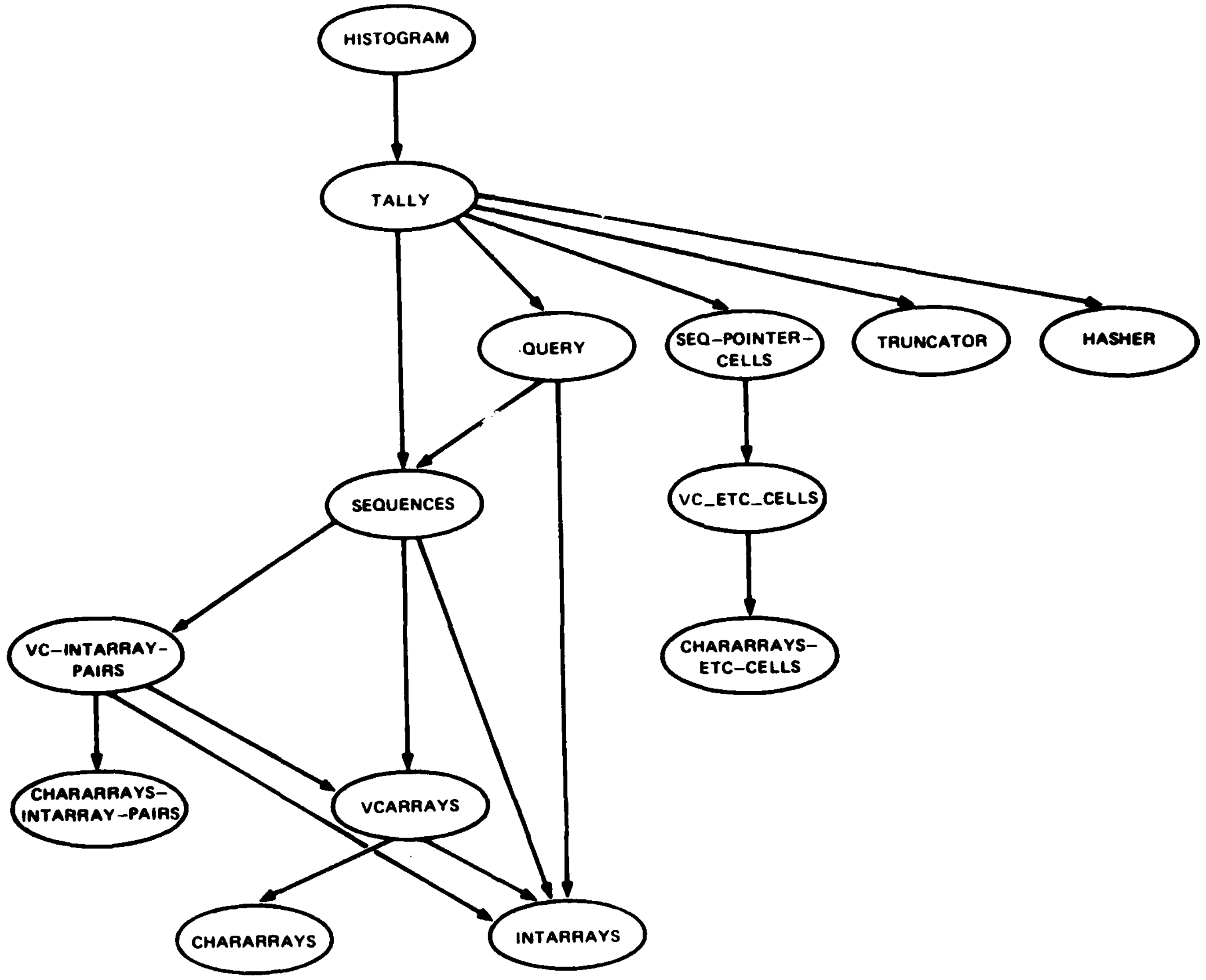
Recall that the "histogram" module provides the operation "hist(n)", which generates the histogram corresponding to a word sequence n. The j-th word and its corresponding frequency count can be retrieved by invoking the operation "getword(j)" and "howmany(j)", respectively.

The module "tally" provides the operations to implement the operations of "histogram". In particular, the implementing operations are:

t_initialize -- identifies a particular word sequence n, for which a histogram is to be formed.

insert_or_increment -- processes the next word of the identified sequence n. If the word has not been previously seen then it is placed in the tables of "tally"; otherwise its associated frequency count is incremented.

Figure VII-2: Structure of Implementation -- Module View



t_retrieve -- returns a word at position j.

t_howmany -- returns the frequency count of the word stored at position j.

reset_tally -- clears the "tally" module.

The major decisions that are revealed in the implementation of "histogram" are as follows:

* The retrieval of a word and its frequency count for the histogram are trivially implemented by the corresponding "tally" operations.

* The resetting of "histogram" is trivially implemented by the "reset_tally" operation.

* The generation of the histogram for sequence n is (approximately) accomplished by a simple program that repeatedly invokes "insert_or_increment" for each successive word of sequence n.

Thus there is no jump in data abstraction between the two modules. The only substantive difference is in procedure abstraction; "hist" processes the words of sequence n in one "fell-swoop", while "insert_or_increment" processes the words one at a time.

Let us now discuss the ILPL implementations as given in Table 22. The header

IMPLEMENTATION histogram IN_TERMS_OF tally

identifies the implemented module and the lower-level ones that implement it. (Note that there may be upper-level implementing modules, too. Only the lower ones, however, appear after the "IN_TERMS_OF".) In general, the implementing set can contain more than one module.

In the TYPES paragraph one can declare named types that will be used in the implementation. As in the case of the representation specifications, the names are merely a shorthand for the type definitions, and thus serve mainly to enhance the readability of the implementation; no new designator types can be declared here.

The EXTERNALREFS paragraph identifies module entities that are referred to in the implementations. In the case of the implemented module ("histogram") all visible operations must be declared, in addition to designator types corresponding to arguments or returned values. Similarly, for the implementing modules (upper and lower), all entities (visible operations, parameters, designator types and scalar types) that are needed in the implementation must be declared.

The IMPLEMENTATIONS paragraph contains the ILPL programs for each of the operations of "histogram". No program is required to initialize the module since its initial state is just the initial state of "tally", transformed by the representation for "histogram" to "tally". Four of the five implementations are almost trivial. We will discuss in detail the program for "getword" to illustrate the notation, and that for "hist", which is the only non-trivial program here.

The program for "getword" is as follows:

```
VPROG getword(INTEGER j) -> truncated_word tw;
   BEGIN
      EXECUTE tw <- t_retrieve(j) THEN
         ON no_string : RAISE(no_word);
         ON NORMAL : RETURN;
      END;
   END;
```

The header line identifies the kind of program (here, VPROG, which means it implements a visible V-function), the program name, and its argument and returned value. The body of the program consists of an invocation of the "tally" operation "t_retrieve". Here a V-function is invoked with the expectation that either an exception will be raised (and handled), or a "normal" return will occur. If the value of j is such as to cause the return of the exception "no_string" -- which, as portrayed in the specification for "tally" (Table 6), corresponds to j being out of bounds -- then the exception "no_word" is raised, and the program terminates. Otherwise, the value of j is acceptable and the result of "t_retrieve(j)" is returned from "getword(j)".

Now consider the program for "hist(n)". This program consists of three parts as follows:

1. Determine if the "tally" tables for the storage of words and frequency counts are cleared; if not raise an exception.

2. Initialize the "tally" module to handle a new sequence n.

3. Process the words of sequence n in turn; if the resources of "tally" are exhausted before all words of the sequence can be successfully processed then raise an exception; otherwise return successfully.

   For the first part we have

```
IF t_len() ~= 0
   THEN RAISE(hist_not_reset);
   END_IF;
```

Here, the V-function "t_len" is invoked for which no exception return is specified -- see the specifications of "t_len" in Table 6 -- and, hence, only the "normal" return need be handled in the implementation. Thus, exception "hist_not_reset" is raised if it has been determined that the value of "t_len" is not 0, which indicates that the "tally" word and frequency tables contain entries, and thus that the histogram, as reflected in terms of its representation, is not in its reset state.

For the second part of the program for "hist(n)" we have

```
t_initialize(n);
```

No exception is specified for this O-function. As indicated in the specifications, the effect here is to initialize the "tally" tables that keep track of the current sequence and the next word in that sequence to be processed.

For the third part we have

```
UNTIL no_more_room DO
   EXECUTE insert_or_increment() THEN
      ON no_more_words :  RETURN;
      ON RESOURCE_ERROR : SIGNAL(no_more_room);
      ON NORMAL : ;
      END;
THEN
   ON no_more_room :
      reset_tally();
      RAISE(RESOURCE_ERROR);
   END;
```

The event "no_more_room" is declared by its position following "UNTIL". Mnemonically, this event is intended to portray the absence of room in the "tally" module to handle additional words. It is intended that the loop body be repeatedly executed until the statement "SIGNAL(no_more_room)" is executed, at which time control passes to the "handler" for the event "no_more_room". Here, the loop body is an EXECUTE statement that involves an invocation of the O-function "insert_or_increment", with three possible results:

1. The exception "no_more_words" is raised, corresponding to all words of the sequence n having been processed. In the event of this exception, we have finished generating the histogram and control can return from the program "hist".

2. The exception "RESOURCE_ERROR" is raised, corresponding to some unspecified exhaustion of resources at a lower level. In the event of this exception, the handler transfers control to the handler for the event "no_more_room". "Reset_tally" is then invoked and causes the "tally" module to be returned to (a state that maps up to) the initial state of the "histogram" module. Following this, the exception "RESOURCE_ERROR" is raised.

3. A "normal" return is made, corresponding to successful processing of the current word.

Note that this program could have been written without the event "no_more_room". The two statements in the handler for this event could have been substituted for the "SIGNAL" statement in the handler for "RESOURCE_ERROR". Our intention was to illustrate the use of declared events, and also to improve the structure and readability of the program.

We will not discuss the remaining programs of the implementation: "howmany", "histlen", and "clearhist". A reader should have no difficulty understanding them.

Now let us proceed to a discussion of the implementation for "tally".

E. Tally Implementation

The concern here is with the implementation of "tally" in terms of the modules "query", "seq_pointer_cells", "hasher", "truncator", and "sequences". For the latter two modules, it is the level5 appearance (the level of "tally") that is referenced in the implementation, as depicted in Figure VII-1. However, for this particular example, the references could be to the module appearances at level4, since there is no partitioning of the level4 appearances of either "sequences" or "truncator" for the use of distinct level5 modules.

As indicated above, "tally" provides the operation "insert_or_increment()", which gives the appearance of processing the next word w in some previously identified sequence n. By "processing", we mean first w is truncated to form a word tw; if tw is not already in the table, it is placed at the next free location of the "t_retrieve" table and the frequency count for that position in the "t_howmany" table is set to 1; otherwise, (tw is already in the table), its count in the "t_howmany" table is incremented by 1. Operations are also provided to access the two tables, and to swap a pair of entries. Recall that the swapping operation permits the rearrangement of the tables once all words of a sequence have been processed.

Now let us briefly review the relevant capabilities of the implementing modules.

The "sequences" module provides the operation "string(n,j)", which permits the retrieval of the j-th word in sequence n.

The "truncator" module provides the parameter "maxlength", the length beyond which characters are ignored in forming a "truncated_word" from a "word".

The "seq_pointer_cells" module provides two cells, one for the storage of a seq designator, and the other for the storage of an integer.

The "hasher" module provides the parametric function

hash(word w; INTEGER upper)

which returns an integer in the range 0..upper-1 corresponding to w.

The "query" module bears most of the burden in implementing "tally". In particular, "query" provides the following:

get_string and check_count, which provide direct access to words and integers, respectively.

size(), which returns the number of words stored in the "get_string" table.

save(w, p), which causes the word w to be saved at the next free position j in the "get_string" table and causes the position p in a "dir" table to hold that value j, provided p is an empty slot in the "dir" table. Also, the value in location j of the "check_count" table is set to 1 if the operation is successfully completed. In essence, "dir" is treated as a hash table, and the argument p is a probe into the table derived from a hashing function on the word w.

add_count(p), which adds 1 to the value in position j of the "check_count" table, where j is the value of "dir(p)".

swap_query(i,j), which simultaneously exchanges the entries in both the "get_string" and "check_count" tables.

reset_query(), which clears the tables to their initial state.

The major decisions reflected by the implementation of "tally" are as follows:

* The initialization of a sequence n involves the storing of n in one of the cells in "seq_pointer_cells" and 0 (pointing to the beginning of the sequence) in the other cell.

* A word is truncated prior to its processing by the "query" operations.

* Reflecting the hash searching scheme, if a word w appears at location j of the "get_string" table, then dir(p) = j for p between hash(w,plen) and some p1, where dir(p1) = 0. A word appears only once in the "get_string" table. These decisions were captured by the invariants in the representation for "tally" (Table 18). Plen, a parameter of "query", is the number of entries in the "dir" table.

Now, let us discuss the interesting implementations as given in Table 23. It is not necessary to discuss the implementations for "t_len", "t_retrieve", "t_howmany", "reset_tally", "t_initialize" and "swap_tally"; they are quite simple.

Let us consider the one interesting implementation, that for "insert_or_increment". First, note the ASSERT statement

```
get_s() ~= ?;
```

It is assumed that before "insert_or_increment" is invoked, a "defined" value, i.e., a seq designator, is in the "get_s" cell of "seq_pointer_cells". Otherwise, it would be necessary to provide an exception corresponding to an "undefined" value being there.

The body of the program can be viewed as consisting of five parts:

1. Initialize local variables n, k to the current seq designator and the index of the last word that was processed.

2. Fetch the next word w from the current sequence. If there is no "next word", then raise the exception "no_more_words".

3. Assign the local variable tw to be the truncation of w.

4. Assign local variable p to be the result of applying "hash" to tw; p is the initial hash probe.

5. Attempt to "save" the word tw. That is, determine if tw is already in the "get_string" table, in which case its corresponding frequency count in the "check_count" table is incremented by one; or if tw is not there, then store it at a "conveniently accessible" location and set its count to 1. The search commences at position hash(tw,plen) of "dir" and continues until one of the following happens: the word is found, an empty slot in the "dir" table at location pi (i.e., dir(pi) = 0) is encountered, or all "dir" slots are examined without success. The attempt to "save" could also fail due to a "resource error".

Let us examine the code for part 5, which corresponds to the following loop statement:

```
FOR p1 FROM p to p + plen - 1 UNTIL done DO
    p2 <- p1 MOD plen;
    EXECUTE save(tw, p2) THEN
       ON hit : add_count(p2);
                SIGNAL(done);
       ON wrong_word : ;
       ON RESOURCE_ERROR : RAISE(RESOURCE_ERROR);
       ON NORMAL : SIGNAL(done);
    END;
THEN
    ON done : store_p(k + 1);
              RETURN;
    ON NORMAL : RAISE(RESOURCE_ERROR);
END;
```

The iteration covers the plen integers starting with the initial hash probe p. The event "done" is declared to correspond to successfully finding tw. Each iteration simply involves an invocation of "save(tw,p2)". There is no need to check explicitly for the exception "no_dir" (see Table 9) since it is clear that p2 will always be between 0 and plen-1. The remaining exceptions and "normal" return associated with "save" are used in an essential way to effect control in the program.

- The exception "hit" corresponds to the word being there. It is thus necessary to increment the count associated with word tw, as is effected by invoking "add_count(p)", and then transferring to the code associated with the handler for event "done". This handler code just causes the index that points to the just processed word of the sequence to be incremented by one. Note that there is no need to check explicitly for the two exceptions of "add_count(p)": "no_dir" and "empty_slot". In this context we know that these exceptions cannot occur.

- The exception "wrong_word" corresponds to a word tw1 ~= tw being stored at location dir(p2) of the "get_string" table. No action is to be taken here, i.e., the next iteration of the loop is to be executed.

- The exception "RESOURCE_ERROR" corresponds, as usual, to an exhaustion of resources at some lower level. In this case the exception "RESOURCE_ERROR" for "insert_or_increment" is raised.

- The "NORMAL" return corresponds to the slot p2 being empty; thus the effects of "save" indicate that tw is now in the "get_string" table at the first empty location, and the corresponding count in the "check_count" table becomes 1. Control is transferred to the "done" handler to update the pointer.

If the loop completes the quota of plen iterations, then control passes to the "NORMAL" handler at the scope of the "FOR" statement. (This handler is optional, and was not needed in the implementation of "hist" -- see previous section.) In this event, all slots of the "dir" table have been investigated without finding the word tw, an empty slot, or without raising an "RESOURCE_ERROR" exception. At this time, it is only possible to raise the exception "RESOURCE_ERROR" for the invocation of "insert_or_increment".

This completes our description of the implementations for "tally". Now, let us proceed downward to the implementation of "query".

F. Query Implementation

The concern here is with the implementation of "query" in terms of "sequences" and "intarrays". As indicated in Figure VII-1, it is the lower level appearance -- level3 -- of "sequences" that is participating in the implementation. Recall that "query" provides two accessible tables: "get_string" and "check_count", which respectively hold words and integers. The primary O-functions are "save(w,p)" and "add_count(p)"; the former enables the storing of word w at the next free position j in the "get_string" table, where the p-th location in the hidden table "dir" is then assigned the value j. Exceptions are returned corresponding to a word w' (possibly being the same as w) already being in the "get_string" table at location dir(p). On the other hand, an invocation of "add_count(p)" causes the j-th location of the "check_count" table, j = dir(p), to be incremented by one provided dir(p)~=0. The other operations of "query" that are to be implemented are "size", "swap_seq", and "reset_query".

Now, let us briefly consider the operations of the two modules used in the implementation. Recall, "sequences" maintains a collection of variable-length word files (sequences). The sequence operations of interest here are:

* create_seq, which creates a new sequence
* string, which enables random retrieval of a word in a sequence
* seqlen, which returns the number of words in a sequence
* append, which attaches a word to the end of a sequence
* clear_seq, which resets a sequence
* swap_seq, which exchanges two words in a sequence.

The module "intarrays" maintains a collection of integer arrays, each of fixed length "leni". The intarray operations of interest here are:

* create_intarray, which creates a new integer array
* getint, which retrieves an integer from a location in the array
* change_int, which causes an identified position in an array to attain a new value.

In the previous chapter we presented the decisions for the representation of "query". Briefly, the "get_string" table is represented by a particular sequence given the name "unique_string". Each of the other tables is represented by an integer array: "check_count" by "count_array", and "dir" by "director_array". In the case of "dir(p)", "defined" values are returned for p in the range 0..plen-1, while in the representation, defined values will appear for positions 1..leni.

The major implementation decisions divulged here are:

* The initialization of "query" creates the sequence and the integer arrays that represent the "query" functions. It is necessary to initialize all positions of the array "director_array" with 0, while "count_array" can have arbitrary values.

* The initialization binds a value to the "query" parameter "plen".

* The successful "saving" of a word appends it to the sequence "unique_string".

* Numerous exception conditions associated with the "sequences" and "intarrays" operations can be shown never to occur, and hence can be omitted from the implementations.

Now let us consider the implementations, as displayed in Table 24. First note the PARAMETERS paragraph, in which designators are declared for two integer arrays and for a sequence. An INITIALIZATION program is required for "query" to create the two integer arrays and the sequence, and to bind the returned designators to the names declared as parameters of the implementation. Also, the module parameter "plen" is bound to the "intarrays" parameter "leni". The binding is accomplished as an assignment statement; for example, the statement

```
unique_string <- create_seq()
```

binds "unique_string" to the designator returned by the invocation of "create_seq()". It is assumed here that these parameters retain their values between invocations of functions in the module. Note that the statement

```
FOR i FROM 1 TO leni DO
   change_int(director_array, i, 0)
```

causes all positions in the "director_array" to be initialized with 0; otherwise, the values would be random integers, as seen from the specifications -- Table 10.

Most of the programs are trivial and do not not require any significant discussion. However, those for "check_count(j) -> v" and "save(w,p)" illustrate a few interesting details. The program body for the former is

```
IF j < 1 OR j > seqlen(unique_string)
   THEN RAISE(no_word);
   END_IF;
v <- getint(count_array, j);
RETURN;
```

In order to determine if j is in bounds it is determined if it is in the range 1..seqlen(unique_string), i.e., if there exists a "defined" word at position j of "unique_string". If j is out-of-bounds the exception "no_word" is returned for "check_count". Otherwise, the value in the j-th position of the "count_array" is returned. Note that no exception is anticipated for the call on "getint". Informally, if it has been determined that there exists a word at position j, then there exists a count for that word in the "count_array". The invariant presented in Table 19:

```
FORALL j | string(unique_string, j) ~= ? :
   j INSET {1 .. leni}
```

captures this property of the usage of "intarrays" and "sequences" by the implementing programs for "query".

Now consider the body for the program for "save(w,p)". It consists of three parts as follows:

1. If p is out of bounds, then the exception "no_dir" is raised.

2. If the p-th slot is empty, then try to append the word w to the "unique_string". If carried out without a resource error, then the "director_array" and the "count_array" are updated.

3. If the p-th slot is not empty, then determine if the word corresponding to it is w (the exception "hit" is raised) or not w (the exception "wrong_word" is raised).

The body of the program is as follows; a blank line separates the three parts.

```
    EXECUTE j <- getint(directorarray, p-1) THEN
       ON no_int : RAISE(no_dir);
       ON NORMAL : ;
    END;

    IF j = 0 THEN
       EXECUTE append(unique_string, w) THEN
          ON RESOURCE_ERROR : RAISE(RESOURCE_ERROR);
          ON NORMAL :
             j1 <- seqlen(unique_string);
             change_int(director_array, p, j1);
             change_int(count_array, j1, 1);
             RETURN;
       END;
    END_IF;

    IF w = string(unique_string, j);
       THEN RAISE(hit);
       ELSE RAISE(wrong_word);
    END_IF;
```

Note that no exception is expected for the two invocations of "change_int" in part two. It has already been determined in part 1 that p is in bounds. Moreover, by the above invariant it is also assured that j1 is between 1 and leni. Again, no exception is expected for the invocation of "string" in part three. An invariant of the "query" representation

```
    FORALL INTEGER p INSET {1 .. leni}
       getint(director_array, p) INSET {0..seqlen(unique_string)}
```

ensures that that value j of part 3 corresponds to an existing word.

This completes our discussion of the implementation for "query". Now let us proceed downward to the implementation for "sequences".

G. Sequences Implementation

The concern here is with the implementation of "sequences" in terms of "vcarrays", "intarrays", and "vc_intarray_pairs". As indicated in Figure VII-1, all of the implementing modules are at the lower level -- level2. Recall that the module "sequences" maintains a collection of variable length word files, here denoted as sequences, where each word is a vector of characters. With each sequence is associated a unique seq designator. The operations provided by the "sequences" module allow direct read access to words in a sequence, append a word to the end of a sequence, and interchange two words of a sequence; a few additional operations, as previously discussed, are also provided.

The "vcarrays" module maintains a collection of variable length character arrays, denoted as vcarrays. With each vcarray we associate a unique "vcarray" designator. The operations of the module needed in the implementation of "sequences" are the following:

char(vca, j) -- returns the j-th character of vcarray vca.

int_for_vcarray(vca) -- returns a unique integer corresponding to vcarray vca.

one_more_char(vca, c) -- appends character c to vcarray vca.

remove_chars(vca, i) -- removes the i last characters from vcarray vca.

clear_vcarray(vca) -- clears vcarray vca to its initial state.

Recall that the module "intarrays" maintains a collection of integer arrays, each of which is of the same fixed length "leni". Each integer array is identified by a unique intarray designator. Operations are provided to enable direct read and write access to a selected integer array.

Let us review the decisions underlying the representation of "sequences" in terms of "vcarrays" and "intarrays". Each sequence is represented by a unique vcarray and a unique integer array. As such, each seq designator is represented as a pair consisting of a vcarray designator and an intarray designator. Each vcarray and intarray designator can appear in the representation of no more than one seq designator. In the representation of a sequence, the vcarray holds the characters of each word in the sequence, but without regard for the separation between words. The characters of a word appear in successive positions, but the groups of characters perceived as words do not necessarily appear in the same order as in the sequence. The information about the separation of characters in the vcarray as words is held in the integer array. In particular, the positions of the first and last characters of the i-th word, as represented in the vcarray, are held in positions 2*i and 2*i - 1 of the integer array. A distinct integer array (with designator "nstrings") holds the current length of each sequence. In particular, the length of sequence n is held in position int_for_vcarray(vca) of nstrings, where vca is the vcarray component of the representation of seq n.

We have not yet indicated the facilities provided by the module "vc_intarray_pairs". For our purposes here, this module provides the operation "create_vc_intarray_pair", which returns a vcarray-intarray pair.

The major decisions expressed in the implementation are the following:

* The creation of a new sequence is done by invoking "create_vc_intarray_pair".

* The initialization of a newly created sequence requires storing 0 in the corresponding location in the "nstrings" array -- a new sequence has zero length.

* The appending of a word to sequence n is implemented by appending the characters, one at a time and in order, to the end of the corresponding vcarray. In addition, the corresponding position in the "nstrings" array is updated to reflect the new length of the sequence, and the corresponding positions in the integer array representing sequence n are updated to indicate the boundary positions of the word.

* The interchanging of two words in sequence n is implemented by interchanging the corresponding boundary positions of the words as represented in the integer array associated with n.

* Numerous exception conditions associated with invocations of "vcarrays" and "intarrays" functions can be shown not to occur, and hence are deleted from the programs.

Now consider the implementations as given in Table 25. This implementation contains the TYPE_MAPPINGS paragraph, which, in general, displays the mapping of the designator types of the implemented module m, essentially as previously given in the representation specification for the machine cluster containing the lowest level appearance of m. For this implementation the designator type seq is mapped to the type vc_intarray_pair, which is defined to be a structure in the TYPES paragraph.

Again, most of abstract programs are readily understood without a detailed discussion. However, we will discuss two of the programs in order to illustrate some of the new syntax introduced here, and to illustrate some of the aspects of abstract programs, in general. First, consider the body of the program for "create_seq() -> n"

```
EXECUTE n <- create_vc_intarray_pair() THEN
   ON RESOURCE_ERROR : RAISE(RESOURCE_ERROR);
   ON NORMAL :
      change_int(nstrings, int_for_vcarray(n.vcarray_part),0);
      RETURN;
   END;
```

Note that in the body, n is a vcarray-intarray pair. The program returns "RESOURCE_ERROR" if no more pairs can be created. Otherwise, it is necessary to insert 0 in the position of the "nstrings" array that corresponds to the newly created sequence. That position is given by

```
int_for_vcarray(n.vcarray_part)
```

Note that no exception is expected for the invocation of "change_int" since the invariant of the representation specification for "sequences"

```
leni >= v_int_bounds
```

guarantees that if a new pair is successfully generated (which implies that a new "vcarray" designator vca is generated) then the value of "int_for_vcarray(vca)" is constrained to not exceed "leni" -- the length of an integer array.

The final statement in the program

```
RETURN;
```

indicates that the returned value is to be the newly generated pair (i.e., n). However, by virtue of the type mapping, the type of the value as perceived by the invoker of "create_seq" is seq. This illustrates the implicit type conversion between the arguments and return value of the operation and the references to them in the body of the program that implements the operation.

Let us now briefly consider the program for "append(n, w)". It consists of four parts as follows:

1. Initialize the local variables vc, i, j, r.

2. Determine if there is sufficient room for one additional word

in the integer array that holds the boundary positions of the words for sequence n. There is room if there are at least two unused positions at the end of the array. A "RESOURCE_ERROR" is raised if two such positions do not exist.

3. For each character of w, in turn, attempt to place it at the end of the vcarray vc that represents n. The appending process stops if there is insufficient room for an additional character, and a "RESOURCE_ERROR" is raised for "append".

4. If parts (2) and (3) are both successfully completed, then update the two next free positions in the "boundary" integer array to correspond to the newly appended word. Also, increment by 1 the position in the "nstrings" array corresponding to sequence n.

With regard to part (3) it is not necessary to remove the characters of w already stored in the vcarray if a resource error is encountered. Although the state of the "vcarrays" module has been changed by the partial appending of a word, the state presented by "sequences" (the state of "vcarrays" and "intarrays" transformed by the representation specifications for "sequences") is not changed, as is proper when an invocation of an operation causes a exception return. Of course, the vcarray vc is now full and cannot accommodate any more characters. As an embellishment, one might replace part (3) by the following program segment

```
FOR k FROM 1 TO LENGTH(w) DO
   EXECUTE one_more_char(vc, w[k]) THEN
      ON RESOURCE_ERROR :
         remove_chars(vc, k-1);
         RAISE(RESOURCE_ERROR);
      ON NORMAL : ;
   END;
END;
```

The k-1 characters of w stored prior to the overflow of the vcarray vc are removed by the invocation of "remove_chars(vc, k-1)", thus returning "vcarrays" to its state prior to the execution of any statements in "append". It is not necessary to check for the "underflow" exception for "remove_chars", since it is clear that there are at least k-1 characters in the vcarray.

This completes our discussion of the implementations. The remainder of the implementations are very straightforward and should be easily comprehended. The reader who has reached this point (without skipping) should indeed be congratulated.

We remind the reader that we have accomplished much more than the design of a single program. Rather, we have specified a family of systems, capable of a large number of implementations, scale choices, and extensions. The example itself is rather simple, but the mechanisms employed are of the same type and nature as those that would be used for the specifications of very large and complex systems. In other words, your effort, we believe, has a great potential application.

A. SPECIFICATIONS FOR THE EXAMPLE

The following tables contain all specifications for the example discussed in this volume.

Table 1: STRUCTURE OF THE EXAMPLE

```
(INTERFACE level6
           (sequences)
           (histogram)
           (truncator))

(INTERFACE level5
           (sequences)
           (tally)
           (truncator))

(INTERFACE level4
           (sequences)
           (seq_pointer_cells)
           (query)
           (hasher)
           (truncator))

(INTERFACE level3
           (sequences)
           (seq_pointer_cells)
           (intarrays)
           (hasher)
           (truncator))

(INTERFACE level2
           (vcarrays)
           (intarrays)
           (vc_intarray_pairs)
           (vc_etc_cells)
           (hasher)
           (truncator))

(INTERFACE level1
           (chararrays)
           (intarrays)
           (chararrays_intarray_pairs)
           (chararrays_etc_cells)
           (hasher)
           (truncator))

(HIERARCHY example
      (level1 IMPLEMENTS level2 USING vcarrays)
      (level2 IMPLEMENTS level3 USING sequences)
      (level3 IMPLEMENTS level4 USING query)
      (level4 IMPLEMENTS level5 USING tally)
      (level5 IMPLEMENTS level6 USING histogram))
```

Table 2: OBJECTS OF MODULES AND MAPPINGS

HISTOGRAM MODULE

VP	VFUN getword(INTEGER j) -> truncated_word w
VP	VFUN howmany(INTEGER j) -> INTEGER i
VD	VFUN histlen() -> INTEGER v
	OFUN hist(seq s)
	OFUN clear_hist()

SEQUENCES MODULE

	seq: DESIGNATOR
VP	VFUN string(seq n; INTEGER j) -> word w
VD	VFUN seqlen(seq n) -> INTEGER v
	OVFUN create_seq() -> seq n
	OFUN clear_seq(seq n)
	OFUN append(seq n; word w)
	OFUN swap_seq(seq n; INTEGER i, j)

TRUNCATOR MODULE

	INTEGER maxlength
VD	truncation(word w) -> truncated_word tw

TALLY MODULE

VP	VFUN t_retrieve(INTEGER j) -> truncated_word tw
VP	VFUN t_howmany(INTEGER j) -> INTEGER v
VD	VFUN t_len() -> INTEGER v
HP	VFUN t_pointer() -> INTEGER v
HP	VFUN t_sequence() -> seq s
	OFUN t_initialize(seq s)
	OFUN insert_or_increment()
	OFUN swap_tally(INTEGER i, j)
	OFUN reset_tally()

SEQ_POINTER_CELLS MODULE

VP	VFUN get_s() -> seq s
VP	VFUN get_p() -> INTEGER v
	OFUN store_s(seq s)

```
        OFUN store_p (INTEGER i)
```

QUERY MODULE

```
        INTEGER plen
VP      VFUN get_string(INTEGER j) -> word w
VP      VFUN check_count(INTEGER j) -> INTEGER v
VD      VFUN size() -> INTEGER v
HP      VFUN dir(INTEGER p) -> INTEGER v
        OFUN save(word w; INTEGER p)
        OFUN add_count(INTEGER p)
        OFUN swap_query(INTEGER i, j)
        OFUN reset_query()
```

HASHER MODULE

```
        INTEGER hash(word w; INTEGER upper)
```

INTARRAYS MODULE

```
        intarray: DESIGNATOR
        INTEGER leni
VP      VFUN getint(intarray m; INTEGER j) -> INTEGER v
        OVFUN create_intarray() -> intarray m
        OFUN change_int(intarray m; INTEGER j, v)
```

VCARRAYS MODULE

```
        vcarray: DESIGNATOR
        INTEGER v_int_bounds
VP      VFUN char(vcarray n; INTEGER i) -> CHAR c
VP      VFUN int_for_vcarray(vcarray n) -> INTEGER v
        OVFUN create_vcarray() -> vcarray n
        OFUN one_more_char(vcarray n; CHAR c)
        OFUN remove_chars(vcarray n; INTEGER i)
        OFUN clear_vcarray(vcarray n)
```

VC_INTARRAY_PAIRS MODULE

```
HP      vc_pair_exists(vc_intarray_pair vnp) -> BOOLEAN b
        OVFUN create_vc_intarray_pair() -> vc_intarray_pair vnp
```

VC_ETC_CELLS MODULE

```
VP      VFUN v_get() -> vcarray vc
VP      VFUN v_get_n() -> intarray n
VP      VFUN v_get_i() -> INTEGER v
        OFUN v_store(vcarray v)
        OFUN v_store_n(intarray n)
        OFUN v_store_i(INTEGER i)
```

CHARARRAYS MODULE

```
        chararray: DESIGNATOR
        INTEGER maxchararrays
        INTEGER lenc
VP      VFUN getchar(chararray n; INTEGER j) -> CHAR c
VP      VFUN int_for_chararray(chararray n) -> INTEGER v
        OVFUN create_chararray() -> chararray n
        OFUN change_char(chararray n; INTEGER j; CHAR c)
```

CHARARRAYS_ETC_CELLS MODULE

```
VP      VFUN c_get() -> chararray nc
VP      VFUN c_get_n() -> intarray n
VP      VFUN c_get_i() -> INTEGER v
        OFUN c_store(chararray nc)
        OFUN c_store_n(intarray n)
        OFUN c_store_i(INTEGER i)
```

CHARARRAYS_INTARRAY_PAIRS MODULE

```
HP      VFUN chararray_pair_exists(chararray_intarray_pair cnp)
            -> BOOLEAN b
        OFUN store_chararray_intarray_pair(chararray_intarray_pair
                                           cnp)
```

HISTOGRAM MAPPING

TALLY MAPPING

QUERY MAPPING

```
seq unique_string
intarray director_array
intarray count_array
```

SEQUENCES MAPPING

```
intarray nstrings
```

VCARRAYS MAPPING

```
intarray length_array
```

Table 3: Sequences Module

```
MODULE sequences
          $( maintains an unspecified number of variable length
           sequences of character strings (words) , each string of
           variable length. For reading, words can be randomly
           accessed. New words can be inserted at the end of a
           sequence. Words can be exchanged)


    TYPES

seq: DESIGNATOR; $( sequences )
word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };


    DEFINITIONS

BOOLEAN no_word(seq n; INTEGER j)
   IS NOT j INSET { 1 .. seqlen(n) };


    FUNCTIONS

VFUN string(seq n; INTEGER j) -> word w;  $( returns the j-th
                                              string in sequence n)
    EXCEPTIONS
       no_word : no_word(n, j);
    INITIALLY
       w = ?;

VFUN seqlen(seq n) -> INTEGER v;
   $( returns the number of strings in sequence n)
    DERIVATION
       CARDINALITY({ INTEGER j | string(n, j) ~= ? });

OVFUN create_seq() -> seq n;
          $( creates a new sequence all words of which are
           undefined. A newly generated designator is returned)
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       n = NEW(seq);

OFUN clear_seq(seq n); $( clears sequence n)
    EFFECTS
       FORALL INTEGER j: 'string(n, j) = ?;

OFUN append(seq n; word w);
   $( appends word w to the end of the sequence n)
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       'string(n, seqlen(n) + 1) = w;

OFUN swap_seq(seq n; INTEGER i, j);
   $( exchanges words in positions i and j of sequence n)
    EXCEPTIONS
       no_word1 : no_word(n, i);
       no_word2 : no_word(n, j);
    EFFECTS
       'string(n, i) = string(n, j);
       'string(n, j) = string(n, i);

END_MODULE
```

Table 4: Truncator Module

```
MODULE truncator  $( provides a function that truncates the length
                  of a word to a fixed maximum length)


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };
truncated_word: { word w | LENGTH(w) <= maxlength };


    PARAMETERS

INTEGER maxlength;


    ASSERTIONS

maxlength > 0;


    FUNCTIONS

VFUN truncation(word w) -> truncated_word tw;
    $( truncates the word w to maxlength)
    DERIVATION
      VECTOR(FOR i FROM 1 TO MIN({ maxlength, LENGTH(w) }):
               w[i]);

END_MODULE
```

Table 5: Histogram Module

```
MODULE histogram
          $( forms and maintains a histogram of the truncated words
           of an identified sequence s; the histogram words are
           stored according to the order of their appearance in the
           sequence s)


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };
truncated_word: { word w | LENGTH(w) <= maxlength };


    DEFINITIONS

BOOLEAN badarg(INTEGER i) IS NOT(i INSET { 1 .. histlen() });

INTEGER occurrences(seq s; truncated_word tw)
   IS CARDINALITY( { 1 .. seqlen(s) }
                  INTER { INTEGER i | truncation(string(s, i))
                                     = tw });
          $( number of occurrences of words whose truncation is tw
           in sequence s)

SET_OF truncated_word occurset(seq s; INTEGER i)
   IS IF i = 0 THEN {}
      ELSE occurset(s,i - 1) UNION {truncation(string(s,i))};
         $( the set of all truncated words whose truncaton occurs
         up to and including the i-th position of sequence s)

truncated_word ith_word(seq s; INTEGER i)
   IS truncation(
        string(s, MIN({ INTEGER j |
                            CARDINALITY(occurset(s, j)) = i})));

                 $(the i-th distinct truncated word in sequence s)
    EXTERNALREFS

    FROM sequences:
seq: DESIGNATOR;
VFUN string(seq s; INTEGER i) -> word w;
VFUN seqlen(seq s) -> INTEGER v;

    FROM truncator:
INTEGER maxlength;
VFUN truncation(word w) -> truncated_word tw;


    FUNCTIONS
```

```
VFUN getword(INTEGER j) -> truncated_word w;
    $( returns the j-th word in the histogram)
    EXCEPTIONS
       no_word : badarg(j);
    INITIALLY
       w = ?;

VFUN howmany(INTEGER j) -> INTEGER v;
          $( returns the number of occurrences of the word at
             position in the histogram)
    EXCEPTIONS
       no_word : badarg(j);
    INITIALLY
       v = ?;

VFUN histlen() -> INTEGER v;
    $(returns the number of words currently stored in the
      histogram)
    DERIVATION
       CARDINALITY({INTEGER i | getword(i) ~= ?});

OFUN hist(seq s); $( forms a histogram of the sequence s)
    EXCEPTIONS
       hist_not_reset :  histlen() ~= 0;
         $( the histogram is not cleared)
       RESOURCE_ERROR;
    EFFECTS
       FORALL INTEGER j:
           'howmany(j) = occurrences(s, 'getword(j))
           AND 'getword(j) = ith_word(s, j);

OFUN clear_hist(); $( clears the histogram)
    EFFECTS
       FORALL INTEGER j: 'getword(j) = ? AND 'howmany(j) = ?;

END_MODULE
```

Table 6: Tally Module

```
MODULE tally
          $( maintains a collection of strings (words) , each of
           whose numbers of occurrences and values can be stored
           and referenced by a unique index. The words are stored
           contiguously. A "swap" function allows the exchanging of
           any two words stored in the histogram)


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };
truncated_word: { word w | LENGTH(w) <= maxlength };


    DEFINITIONS

BOOLEAN no_string(INTEGER j) IS j < 1 OR j > t_len();

truncated_word truncation(word w)
   IS VECTOR(FOR i FROM 1 TO MIN({ maxlength, LENGTH(w) })
            : w[i]);


    EXTERNALREFS

    FROM sequences:
seq: DESIGNATOR;
VFUN string(seq n; INTEGER i) -> word w;
VFUN seqlen(seq n) -> INTEGER v;

    FROM truncator:
INTEGER maxlength;


    FUNCTIONS

VFUN t_retrieve(INTEGER j) -> truncated_word tw;
     $( returns j-th string in tally which has one or more
       occurrences)
    EXCEPTIONS
       no_string : no_string(j);
    INITIALLY
       tw = ?;

VFUN t_howmany(INTEGER j) -> INTEGER v;
   $( returns no. of occurrences of the j-th non-undefined
      string in tally)
    EXCEPTIONS
       no_string : no_string(j);
    INITIALLY
```

```
      v = ?;

VFUN t_len() -> INTEGER v;  $( returns current no. of strings in
                             tally)
    DERIVATION
      CARDINALITY({ INTEGER i | t_howmany(i) ~= ? });

VFUN t_pointer() -> INTEGER v;
          $( returns the value of the pointer into the sequence
           whose histogram is being computed)
    HIDDEN;
    INITIALLY
      v = ?;

VFUN t_sequence() -> seq s;  $( returns the current sequence
                                 whose histogram is being computed)
    HIDDEN;
    INITIALLY
      s = ?;

OFUN t_initialize(seq s);  $( sets current sequence to be s, and
                               the pointer to 0)
    EFFECTS
      't_sequence() = s;
      't_pointer() = 0;

OFUN insert_or_increment();
          $( inserts the next string of the current sequence at the
           end of the tally sequence, provided that string has not
           been previously stored; sets the count value for that
           string to 1; if the word has been previously stored then
           its count-value is incremented by 1)
    ASSERTIONS
      t_sequence() ~= ?;
    EXCEPTIONS
      no_more_words : t_pointer() >= seqlen(t_sequence());
      RESOURCE_ERROR;
    EFFECTS
      't_pointer() = t_pointer() + 1;
      LET INTEGER i |
         t_retrieve(i)
          = truncation(string(t_sequence(), 't_pointer()))
       IN IF i ~= ?
           THEN 't_howmany(i) = t_howmany(i) + 1
           ELSE    't_retrieve(t_len() + 1)
                 = truncation(string(t_sequence(), 't_pointer()))
             AND 't_howmany(t_len() + 1) = 1;

OFUN swap_tally(INTEGER i, j);  $( exchanges the truncated words
                                  and the tallies of indices i and j)
    EXCEPTIONS
      no_string1 : no_string(i);
```

```
        no_string2 : no_string(j);
     EFFECTS
        't_retrieve(i) = t_retrieve(j);
        't_retrieve(j) = t_retrieve(i);
        't_howmany(i) = t_howmany(j);
        't_howmany(j) = t_howmany(i);

OFUN reset_tally(); $( resets the set of strings to initial state)
     EFFECTS
        't_pointer() = 0;
        FORALL INTEGER i:
           't_howmany(i) = ? AND 't_retrieve(i) = ?;

END_MODULE
```

Table 7: Hasher Module

```
MODULE hasher
          $( provides a mechanism for returning an integer
           (hash probe) corresponding to a word. The range of the
           returned integer is between 0 and "upper" -1)


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };


    PARAMETERS

INTEGER hash(word w; INTEGER upper);


    ASSERTIONS

FORALL word w; INTEGER j:
   IF j < 1
     THEN hash(w, j) = ?
     ELSE hash(w, j) INSET { 0 .. j - 1 };

END_MODULE
```

Table 8: Seq_Pointer_Cells Module

```
MODULE seq_pointer_cells  $( maintains two cells, one for a
                           sequence and one for a pointer)


    EXTERNALREFS

    FROM sequences:
seq: DESIGNATOR;


    FUNCTIONS

VFUN get_s() -> seq s; $( returns the stored seq designator)
    ASSERTIONS
       get_s() ~= ?;
    INITIALLY
       s = ?;

VFUN get_p() -> INTEGER v;  $( returns the value of the integer
                            pointer)
    INITIALLY
       v = 0;

OFUN store_s(seq s); $( stores a sequence designator)
    EFFECTS
       'get_s() = s;

OFUN store_p(INTEGER i); $( stores the INTEGER i)
    EFFECTS
       'get_p() = i;

END_MODULE
```

Table 9: Query Module

```
MODULE query
          $( maintains a sequence of strings (words) that are
           referenced by an index and by auxiliary pointers
           (directors -- i.e. hash table indices) and retains a
           count of the number of occurrences of the string
           associated with each index. A new word can be
           arbitrarily inserted at the end of the query sequence
           and associated with a specified pointer. Two entries in
           the query sequence can also be swapped.)


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };


    PARAMETERS

INTEGER plen; $( total number of directors)


    DEFINITIONS

BOOLEAN badarg(INTEGER j) IS NOT j INSET { 1 .. size() };

BOOLEAN baddir(INTEGER p) IS NOT p INSET { 0 .. (plen - 1) };


    ASSERTIONS

plen >= 0;


    FUNCTIONS

VFUN get_string(INTEGER j) -> word w; $( returns the j-th string)
    EXCEPTIONS
       no_word : badarg(j);
    INITIALLY
       w = ?;

VFUN check_count(INTEGER j) -> INTEGER v;
   $( returns current no. of occurrences of j-th string)
    EXCEPTIONS
       no_word : badarg(j);
    INITIALLY
       v = ?;

VFUN size() -> INTEGER v; $( returns current no. of strings)
    DERIVATION
```

```
        CARDINALITY({ INTEGER j | get_string(j) ~= ? })';

VFUN dir(INTEGER p) -> INTEGER v;
          $( returns the p-th director to the string sequence and
           frequency lists; p is a hash table index)
    HIDDEN;
    INITIALLY
       v =(IF p INSET { 0 .. plen - 1 } THEN 0 ELSE ?);

OFUN save(word w; INTEGER p);
          $( stores a word at the end of the query sequence,
           provided the pth director does not point to w or any
           other word. The pth director is set to point to the
           newly stored string.)
    EXCEPTIONS
       no_dir : baddir(p);
       hit : get_string(dir(p)) = w;
       wrong_word : dir(p) ~= 0;
       RESOURCE_ERROR;
    EFFECTS
       'get_string(size() + 1) = w;
       'dir(p) = size() + 1;
       'check_count(size() + 1) = 1;

OFUN add_count(INTEGER p);  $( increments the count of the string
                            associated with the pth director by one)
    EXCEPTIONS
       no_dir : baddir(p);
       empty_slot : dir(p) = 0;
    EFFECTS
       'check_count(dir(p)) = check_count(dir(p)) + 1;

OFUN swap_query(INTEGER i, j);  $( exchanges the words at positions
                                 i and j of the query sequence)
    EXCEPTIONS
       no_word1 : badarg(i);
       no_word2 : badarg(j);
    EFFECTS
       'get_string(i) = get_string(j);
       'get_string(j) = get_string(i);
       'check_count(i) = check_count(j);
       'check_count(j) = check_count(i);

OFUN reset_query(); $( resets the module)
    EFFECTS
       FORALL INTEGER j: 'check_count(j) = ?;
       FORALL INTEGER j: 'get_string(j) = ?;
       FORALL INTEGER p INSET { 0 .. plen - 1 }: 'dir(p) = 0;

END_MODULE
```

Table 10: Intarrays Module

```
MODULE intarrays  $( maintains a fixed, but unspecified, number of
                    fixed-length integer arrays)


    TYPES

intarray: DESIGNATOR;


    PARAMETERS

INTEGER leni; $( no. of integers in an array)


    DEFINITIONS

BOOLEAN no_int(INTEGER j) IS NOT j INSET { 1 .. leni };


    FUNCTIONS

VFUN getint(intarray m; INTEGER j) -> INTEGER v;
   $( returns j-th integer in array m)
    EXCEPTIONS
       no_int : no_int(j);
    INITIALLY
       v = ?;

OVFUN create_intarray() -> intarray m;
          $( creates a new intarray whose elements are initialized
           to some defined integer; a resource error is returned if
           no more intarrays can be created)
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       m = NEW(intarray);
       FORALL INTEGER j INSET { 1 .. leni }:
          'getint(m, j) =(SOME INTEGER i | i ~= ?);

OFUN change_int(intarray m; INTEGER j, v);  $( replaces j-th integer
                                              in array m by v)
    EXCEPTIONS
       no_int : no_int(j);
    EFFECTS
       'getint(m, j) = v;

END_MODULE
```

Table 11: Vcarrays Module

```
MODULE vcarrays
          $( maintains a unspecified number of variable-length
           character strings i.e., vcarrays. Each vcarray has a
           current length within which any character can be
           randomly accessed. The vcarray is modified by appending
           a character to the end of the vcarray, removing a string
           of characters from the end of the vcarray, or by
           clearing the vcarray.)


    TYPES

vcarray: DESIGNATOR;  $( names for variable-length character arrays)

    PARAMETERS

INTEGER v_int_bounds; $( max value of integer corresponding to a
                         vcarray designator)


    DEFINITIONS

INTEGER vclen(vcarray n) IS
    CARDINALITY({ INTEGER j | char(n, j) ~= ? });
   $( returns no. of characters in character vcarray n)


    FUNCTIONS

VFUN char(vcarray n; INTEGER i) -> CHAR c;
   $( returns j-th character in vcarray n)
    EXCEPTIONS
       no_char : NOT i INSET { 1 .. vclen(n) };
    INITIALLY
       c = ?;

VFUN int_for_vcarray(vcarray n) -> INTEGER v;
   $( returns a unique integer for each vcarray)
    INITIALLY
       v = ?;

OVFUN create_vcarray() -> vcarray n;
          $( creates a new vcarray the contents of which are
           undefined. A newly generated designator is returned.
           A unique integer is associated with this designator.)

    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       n = NEW(vcarray);
```

```
        'int_for_vcarray(n)
             = (SOME INTEGER i | i INSET { 1 .. v_int_bounds } AND
                (FORALL vcarray n1 ~= n : int_for_vcarray(n1) ~= i));

OFUN one_more_char(vcarray n; CHAR c);  $( adds character to end of
                                       vcarray n)
     EXCEPTIONS
        RESOURCE_ERROR;
     EFFECTS
        'char(n, vclen(n) + 1) = c;

OFUN remove_chars(vcarray n; INTEGER i);  $( removes i characters
                                          from the end of vcarray n)
     EXCEPTIONS
        underflow : vclen(n) < i;
     EFFECTS
        FORALL INTEGER j INSET { 1 .. i }:
           'char(n, vclen(n) - j + 1) = ?;

OFUN clear_vcarray(vcarray n);  $( resets vcarray to
                                the empty state)
     EFFECTS
        FORALL INTEGER j: 'char(n, j) = ?;

END_MODULE
```

Table 12: Vc_Intarray_Pairs Module

```
MODULE vc_intarray_pairs
          $( stores pairs, each composed of a vcarray designator
           and an intarray designator. Each pair is the
           representation of a sequence designator)


    TYPES

vc_intarray_pair: STRUCT_OF(vcarray vcarray_part;
                            intarray intarray_part);


    EXTERNALREFS

    FROM vcarrays:
vcarray: DESIGNATOR;
OVFUN create_vcarray() -> vcarray vc;

    FROM intarrays:
intarray: DESIGNATOR;
OVFUN create_intarray() -> intarray n;


    FUNCTIONS

VFUN vc_pair_exists(vc_intarray_pair vnp) -> BOOLEAN b;
          $( returns TRUE if the pair vnp has been previously
           stored)
    HIDDEN;
    INITIALLY
       b = FALSE;

OVFUN create_vc_intarray_pair() -> vc_intarray_pair vnp;
          $( creates a new pair by creating a new vcarray and
           intarray, and joining them as a pair (STRUCT))
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       vnp
          = STRUCT(EFFECTS_OF create_vcarray(),
                   EFFECTS_OF create_intarray());
       'vc_pair_exists(vnp) = TRUE;

END_MODULE
```

Table 13: Vc_Etc_Cells Module

```
MODULE vc_etc_cells
          $( provides separate cells for the storage of vcarray
           designators, intarray designators, and integers)


    EXTERNALREFS

    FROM vcarrays:
vcarray: DESIGNATOR;

    FROM intarrays:
intarray: DESIGNATOR;


    FUNCTIONS

VFUN v_get() -> vcarray vc;  $( returns a stored vcarray designator)
    ASSERTIONS
       v_get() ~= ?;
    INITIALLY
       vc = ?;

VFUN v_get_n() -> intarray n;  $( returns a stored intarray
                                  designator)
    ASSERTIONS
       v_get_n() ~= ?;
    INITIALLY
       n = ?;

VFUN v_get_i() -> INTEGER v; $( returns a stored integer)
    INITIALLY
       v = 0;

OFUN v_store(vcarray vc); $( stores a vcarray designator)
    EFFECTS
       'v_get() = vc;

OFUN v_store_n(intarray n); $( stores a intarray designator)
    EFFECTS
       'v_get_n() = n;

OFUN v_store_i(INTEGER i); $( stores an integer)
    EFFECTS
       'v_get_i() = i;

END_MODULE
```

Table 14: Chararrays Module

```
MODULE chararrays  $( maintains a fixed number of fixed-length
                    character arrays)


    TYPES

chararray: DESIGNATOR;


    PARAMETERS

INTEGER maxchararrays;  $( the maximum number of chararrays that
                         the module can support)
INTEGER lenc; $( the length of each chararray)


    DEFINITIONS

BOOLEAN no_char(INTEGER j) IS NOT j INSET { 1 .. lenc };

BOOLEAN too_many_chararrays
   IS CARDINALITY({ chararray n | getchar(n, 1) ~= ? })
     >= maxchararrays;


    ASSERTIONS

lenc >= 1;


    FUNCTIONS

VFUN getchar(chararray n; INTEGER j) -> CHAR c;
   $( returns j-th character in n-th array)
    EXCEPTIONS
       no_char : no_char(j);
    INITIALLY
       c = ?;

VFUN int_for_chararray(chararray n) -> INTEGER v;
          $( returns the unique integer corresponding to chararray
           n)
    INITIALLY
       v = ?;

OVFUN create_chararray() -> chararray n;
          $( creates a new chararray the contents of which become
           some defined (not undefined) characters. A newly
           generated designator is returned)
    EXCEPTIONS
```

```
        too_many_chararrays : too_many_chararrays;
     EFFECTS
        n = NEW(chararray);
         'int_for_chararray(n)
           = (SOME INTEGER i | i INSET { 1 .. maxchararrays } AND
              (FORALL chararray n1 ~= n : int_for_chararray(n1) ~= i));
        FORALL INTEGER j INSET { 1 .. lenc }:
           'getchar(n, j) = (SOME CHAR c | c ~= ?);

OFUN change_char(chararray n; INTEGER j; CHAR c);
   $( replaces j-th character in n-th array by c)
     EXCEPTIONS
        no_char : no_char(j);
     EFFECTS
        'getchar(n, j) = c;

END_MODULE
```

Table 15: Chararrays_Intarray_Pairs Module

```
MODULE chararrays_intarray_pairs
          $( stores pairs, each composed of a chararray designator
             and an intarray designator. Each
             pair is the representation of a vc_intarray_pair)


    TYPES

chararray_intarray_pair:
  STRUCT_OF(chararray chararray_part; intarray intarray_part);


    EXTERNALREFS

    FROM chararrays:
chararray: DESIGNATOR;

    FROM intarrays:
intarray: DESIGNATOR;


    FUNCTIONS

VFUN chararray_pair_exists(chararray_intarray_pair cnp)
                              -> BOOLEAN b;
   $( returns TRUE if the pair cnp has been previously stored)
    HIDDEN;
    INITIALLY
       b = FALSE;

OFUN store_chararray_intarray_pair(chararray_intarray_pair cnp);
          $( creates a new pair by creating a new chararray and
            intarray, and joining them as a pair (STRUCT))
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       'chararray_pair_exists(cnp) = TRUE;

END_MODULE
```

Table 16: Chararrays_Etc_Cells Module

```
MODULE chararrays_etc_cells
          $( provides separate cells for the storage of chararray
           designators, intarray designators, and integers)

    EXTERNALREFS

    FROM chararrays:
chararray: DESIGNATOR;

    FROM intarrays:
intarray: DESIGNATOR;

    FUNCTIONS

VFUN c_get() -> chararray nc;
   $( returns a stored chararray designator)
    ASSERTIONS
       c_get() ~= ?;
    INITIALLY
       nc = ?;

VFUN c_get_n() -> intarray n;  $( returns a stored intarray
                                  designator)
    ASSERTIONS
       c_get() ~= ?;
    INITIALLY
       n = ?;

VFUN c_get_i() -> INTEGER v; $( returns a stored integer)
    INITIALLY
       v = 0;

OFUN c_store(chararray nc); $( stores a chararray designator)
    EFFECTS
       'c_get() = nc;

OFUN c_store_n(intarray n); $( stores a intarray designator)
    EFFECTS
       'c_get_n() = n;

OFUN c_store_i(INTEGER i); $( stores an integer)
    EFFECTS
       'c_get_i() = i;

END_MODULE
```

Table 17: Histogram Mapping

```
MAP histogram TO tally;


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };
truncated_word: { word w | LENGTH(w) <= maxlength };


    EXTERNALREFS

    FROM histogram:
VFUN getword(INTEGER i) -> truncated_word tw;
VFUN howmany(INTEGER i) -> INTEGER v;

    FROM truncator:
INTEGER maxlength;

    FROM tally:
VFUN t_retrieve(INTEGER i) -> truncated_word tw;
VFUN t_howmany(INTEGER i) -> INTEGER v;


    MAPPINGS


getword(INTEGER i): t_retrieve(i);

howmany(INTEGER i): t_howmany(i);

END_MAP
```

Table 18: Tally Mapping

```
MAP tally
     TO query, hasher, seq_pointer_cells;


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };
truncated_word: { word w | LENGTH(w) < maxlength };


    DEFINITIONS

BOOLEAN probe_succeeds(word w)
   IS EXISTS INTEGER p INSET { 0 .. plen - 1 }:
         get_string(dir(p)) = w
        AND (FORALL INTEGER i INSET
              { hash(w, plen) ..
                  IF p < hash(w, plen)
                      THEN p + plen ELSE p }:
              dir(i MOD plen) ~= 0);
          $( returns TRUE if for word w there is a pointer p to the
           word and that a linear search from the hash address of w
           will hit the entry with w in it--that is, not hit an
           empty entry first)


    EXTERNALREFS

    FROM tally:
VFUN t_retrieve(INTEGER i) -> truncated_word w;
VFUN t_howmany(INTEGER i) -> INTEGER v;
VFUN t_sequence() -> seq s;
VFUN t_pointer() -> INTEGER p;

    FROM query:
INTEGER plen;
VFUN get_string(INTEGER i) -> word w;
VFUN check_count(INTEGER i) -> INTEGER v;
VFUN dir(INTEGER p) -> INTEGER v;

    FROM hasher:
INTEGER hash(word w; INTEGER upper);

    FROM sequences:
seq: DESIGNATOR;

    FROM seq_pointer_cells:
VFUN get_s() -> seq s;
VFUN get_p() -> INTEGER p;
```

```
    FROM truncator:
INTEGER maxlength;


    INVARIANTS

FORALL word w ~= ?:
   CARDINALITY({ INTEGER j | get_string(j) = w }) <= 1;
          $( guarantees that all defined words are stored no more
           that once)

FORALL word w | w ~= ? AND(EXISTS INTEGER j: get_string(j) = w):
   probe_succeeds(w);
   $( guarantees that all defined words that are stored in
   "get_string" possess an appropriate link to their position in
   "get_string")


    MAPPINGS


t_retrieve(INTEGER i): get_string(i);

t_howmany(INTEGER i): check_count(i);

t_sequence(): get_s();

t_pointer(): get_p();

END_MAP
```

Table 19: Query Mapping

```
MAP query, sequences TO sequences, intarrays;


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };
seq1: { seq n | n ~= unique_string };


    PARAMETERS

seq unique_string;  $( the unique string sequence; stores the
                       words of query)
intarray director_array;  $( the integer array containing the
                             directors into the unique sequence)
intarray count_array;  $( the integer array containing the counts for
                          each word in the unique sequence)


    DEFINITIONS

INTEGER seqlen(seq n) IS
     CARDINALITY({ INTEGER j | string(n, j) ~= ?});
   $( number of words in sequence n)


    EXTERNALREFS

    FROM sequences:
seq: DESIGNATOR;
VFUN string(seq n; INTEGER j) -> word w;

    FROM query:
INTEGER plen;
VFUN dir(INTEGER p) -> INTEGER i;
VFUN check_count(INTEGER j) -> INTEGER v;
VFUN get_string(INTEGER j) -> word w;

    FROM intarrays:
intarray: DESIGNATOR;
INTEGER leni;
VFUN getint(intarray m; INTEGER j) -> INTEGER v;


    INVARIANTS

FORALL INTEGER j INSET { 1 .. leni }:
   getint(director_array, j) INSET { 0 .. seqlen(unique_string) };
          $( means that all integers in the director_array are
           between 0 and the length of the unique_string sequence.
```

```
                This invariant guarantees that any number retrieved from
                the director array (and within the designated bounds)
                either corresponds to a null entry (0) or indexes a valid
                entry in the unique sequence)

CARDINALITY( {INTEGER j | getint(director_array, j) = 0 })
     = leni - seqlen(unique_string);
                $( guarantees that if there is an empty slot (a zero
                entry) in the director array, then there are fewer than
                leni words in the unique sequence; also guarantees that
                leni is no less than the length of the unique sequence
                (so that any reference to this sequence will also be a
                valid reference into the count array))


    MAPPINGS

               $( the first two mappings are for sequence entities,
                indicating that all sequence designators are available
                at the upper interface with the exception of
                "unique_string" , used exclusively by query)

seq: seq1;

string(seq n; INTEGER j): string(n, j);

   $( the remaining mappings are for query primitives)

plen: leni;

get_string(INTEGER j): string(unique_string, j);

dir(INTEGER p): getint(director_array, p + 1);

check_count(INTEGER j): getint(count_array, j);

END_MAP
```

Table 20: Sequences Mapping

```
MAP sequences, intarrays, seq_pointer_cells
     TO vcarrays, intarrays, vc_intarray_pairs, vc_etc_cells;


    TYPES

intarray1:
  { intarray m | NOT m INSET(inclength UNION { nstrings }) };
vc_intarray_pair:
  STRUCT_OF(vcarray vcarray_part; intarray intarray_part);
vc_intarray_pair1:
  { vc_intarray_pair vcnp | vc_pair_exists(vcnp) = TRUE };
word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };


    PARAMETERS

intarray nstrings;  $( integer array storing the number of strings
                     in the sequences)


    DEFINITIONS

INTEGER len_seq(vc_intarray_pair1 vnp)
   IS getint(nstrings, int_for_vcarray(vnp.vcarray_part));
          $( mapped length of sequence n)

INTEGER ival(vc_intarray_pair1 vnp; INTEGER i)
   IS getint(vnp.intarray_part, i);  $( value in the intarray that
                                     holds the word boundary
                                     positions for sequence n)

SET_OF intarray inclength
   IS {intarray m | EXISTS vc_intarray_pair vnp : vc_pair_exists(vnp)
                    AND vnp.intarray_part = m};
          $( integer arrays, each of which corresponds to a
           vcarray; the values in the positions of each array are
           the vcarray positions at the boundaries of strings)


    EXTERNALREFS

    FROM sequences:
seq: DESIGNATOR;
VFUN string(seq n; INTEGER j) -> word w;

    FROM intarrays:
intarray: DESIGNATOR;
INTEGER leni;
VFUN getint(intarray m; INTEGER j) -> INTEGER v;
```

```
    FROM seq_pointer_cells:
VFUN get_s() -> seq n;
VFUN get_p() -> INTEGER p;

    FROM vcarrays:
vcarray: DESIGNATOR;
INTEGER v_int_bounds;
VFUN char(vcarray vc; INTEGER j) -> CHAR c;
VFUN int_for_vcarray(vcarray vc) -> INTEGER v;

    FROM vc_etc_cells:
VFUN v_get() -> vcarray vc;
VFUN v_get_n() -> intarray n;
VFUN v_get_i() -> INTEGER v;

    FROM vc_intarray_pairs:
VFUN vc_pair_exists(vc_intarray_pair vcnp) -> BOOLEAN b;


    INVARIANTS

v_int_bounds <= leni;
      $( guarantees that any integer corresponding to a vcarray is a
      valid index in an integer array)


    MAPPINGS

               $( the first three mappings are for the
               representation of the intarray primitives)

intarray: intarray1;

leni: leni;

getint(intarray m; INTEGER j): getint(m, j);

          $( the next two mappings are for the representation of
           string primitives)

seq: vc_intarray_pair1;

string(seq n; INTEGER j):
    IF j INSET { 1 .. len_seq(n) }
      THEN VECTOR(FOR i FROM 0
                        TO ival(n, 2*j) - ival(n, 2*j - 1) :
                     char(n.vcarray_part,
                          i + ival(n, 2*j - 1)))
      ELSE ?;

                $( the following mappings are for seq_pointer_cells
                    primitives)
get_p(): v_get_i();
get_s(): STRUCT(v_get(), v_get_n());
END_MAP
```

Table 21: Vcarrays Mapping

```
MAP vcarrays, intarrays, vc_intarray_pairs, vc_etc_cells
   TO chararrays, intarrays, chararrays_intarray_pairs,
      chararrays_etc_cells;


    TYPES

intarray1: { intarray m | m ~= length_array };
vc_intarray_pair:
  STRUCT_OF(vcarray vcarray_part; intarray intarray_part);
chararray_intarray_pair:
  STRUCT_OF(chararray chararray_part; intarray intarray_part);


    PARAMETERS

intarray length_array;  $( the designator for the integer array that
                           contains the lengths of the vcarrays)


    EXTERNALREFS

    FROM vcarrays:
vcarray: DESIGNATOR;
INTEGER v_int_bounds;
VFUN char(vcarray n; INTEGER j) -> CHAR c;
VFUN int_for_vcarray(vcarray vc) -> INTEGER i;

    FROM vc_intarray_pairs:
VFUN vc_pair_exists(vc_intarray_pair vnp) -> BOOLEAN b;

    FROM vc_etc_cells:
VFUN v_get() -> vcarray vc;
VFUN v_get_n() -> intarray m;
VFUN v_get_i() -> INTEGER v;

    FROM intarrays:
intarray: DESIGNATOR;
INTEGER leni;
VFUN getint(intarray m; INTEGER i) -> INTEGER v;

    FROM chararrays:
chararray: DESIGNATOR;
INTEGER maxchararrays;
VFUN getchar(chararray n; INTEGER i) -> CHAR c;
VFUN int_for_chararray(chararray n) -> INTEGER v;

    FROM chararrays_intarray_pairs:
VFUN chararray_pair_exists(chararray_intarray_pair cnp)
                              -> BOOLEAN b;
    FROM chararrays_etc_cells:
VFUN c_get() -> chararray n;
VFUN c_get_n() -> intarray m;
VFUN c_get_i() -> INTEGER v;


    INVARIANTS

leni >= maxchararrays;


    MAPPINGS

   $( the first four mappings are for vcarray primitives)

vcarray: chararray;

v_int_bounds: leni;


char(vcarray n; INTEGER j):
    IF j INSET { 1 .. getint(length_array, int_for_chararray(n)) }
      THEN getchar(n, j)
      ELSE ?;

int_for_vcarray(vcarray n): int_for_chararray(n);

          $( the following three mappings are for intarray
           primitives)

intarray: intarray1;

leni: leni;

getint(intarray m; INTEGER i): getint(m, i);

   $( the following three mappings are for the vc_etc_cells
   primitives)

v_get(): c_get();

v_get_n(): c_get_n();

v_get_i(): c_get_i();

   $( the following mapping is for the vc_intarray_pairs
   module)

vc_pair_exists(vc_intarray_pair vnp): chararray_pair_exists(vnp);

END_MAP
```

Table 22: Histogram Implementation

```
IMPLEMENTATION histogram IN_TERMS_OF tally;


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };
truncated_word: { word w | LENGTH(w) <= maxlength} ;


    EXTERNALREFS

    FROM histogram:
VFUN getword(INTEGER i) -> truncated_word w;
VFUN howmany(INTEGER i) -> INTEGER v;
VFUN histlen() -> INTEGER v;
OFUN hist(seq s);
OFUN clear_hist();

    FROM tally:
VFUN t_retrieve(INTEGER j) -> word w;
VFUN t_howmany(INTEGER j) -> INTEGER v;
VFUN t_len() -> INTEGER v;
OFUN t_initialize(seq s);
OFUN insert_or_increment();
OFUN reset_tally();

    FROM sequences:
seq: DESIGNATOR;

    FROM truncator:
INTEGER maxlength;


    IMPLEMENTATIONS


VPROG getword(INTEGER j) -> truncated_word tw;
BEGIN
   EXECUTE tw <- t_retrieve(j) THEN
     ON no_string : RAISE(no_word);
     ON NORMAL: RETURN;
   END;
END;

VPROG howmany(INTEGER j) -> INTEGER v;
BEGIN
   EXECUTE v <- t_howmany(j) THEN
     ON no_string : RAISE(no_word);
     ON NORMAL : RETURN;
   END;
```

```
END;

VPROG histlen() -> INTEGER v;
BEGIN
   v <- t_len();
   RETURN;
END;

OPROG hist(seq n);
BEGIN
   IF t_len() ~= 0
      THEN RAISE(hist_not_reset);
      END_IF;
   t_initialize(n);
   UNTIL no_more_room DO
      EXECUTE insert_or_increment() THEN
         ON no_more_words :
            RETURN();
         ON RESOURCE_ERROR : SIGNAL(no_more_room);
         ON NORMAL : ;
         END;
      THEN
      ON no_more_room :
            reset_tally();
            RAISE(RESOURCE_ERROR);
   END;
END;

OPROG clearhist();
BEGIN
   reset_tally();
END;


END_IMPLEMENTATION
```

Table 23: Tally Implementation

```
IMPLEMENTATION tally IN_TERMS_OF query, sequences, hasher,
                                seq_pointer_cells, truncator;


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };
truncated_word: { word w | LENGTH(w) <= maxlength };


    EXTERNALREFS

    FROM tally:
VFUN t_retrieve(INTEGER j) -> truncated_word w;
VFUN t_howmany(INTEGER j) -> INTEGER v;
VFUN t_len() -> INTEGER v;
OFUN t_initialize(seq s);
OFUN insert_or_increment();
OFUN swap_tally(INTEGER i, j);
OFUN reset_tally();

    FROM query:
INTEGER plen;
VFUN get_string(INTEGER j) -> word w;
VFUN check_count(INTEGER j) -> INTEGER v;
VFUN size() -> INTEGER v;
OFUN save(word w; INTEGER p);
OFUN add_count(INTEGER p);
OFUN swap_query(INTEGER i, j);
OFUN reset_query();

    FROM sequences:
seq: DESIGNATOR;
VFUN string(seq n; INTEGER j) -> word w;

    FROM hasher:
INTEGER hash(word w; INTEGER upper);

    FROM seq_pointer_cells:
VFUN get_p() -> INTEGER i;
VFUN get_s() -> seq n;
OFUN store_p(INTEGER i);
OFUN store_s(seq n);

    FROM truncator:
INTEGER maxlength;
VFUN truncation(word w) -> truncated_word tw;


    IMPLEMENTATIONS
```

```
VPROG t_len() -> INTEGER v;
BEGIN
   v <- size();
   RETURN;
END;

VPROG t_retrieve(INTEGER j) -> truncated_word tw;
BEGIN
   EXECUTE tw <- get_string(j) THEN
      ON no_word : RAISE(no_string);
      ON NORMAL : RETURN;
      END;
END;

VPROG t_howmany(INTEGER j) -> INTEGER v;
BEGIN
   EXECUTE v <- check_count(j) THEN
      ON no_word : RAISE(no_string);
      ON NORMAL : RETURN;
      END;
END;

OPROG reset_tally();
BEGIN
   reset_query();
END;

OPROG t_initialize(seq n);
BEGIN
   store_s(n);
   store_p(0);
END;

OPROG swap_tally(INTEGER i,j);
BEGIN
   EXECUTE swap_query(i, j) THEN
      ON no_word1 : RAISE(no_string1);
      ON no_word2 : RAISE(no_string2);
      ON NORMAL : RETURN;
      END;
END;

OPROG insert_or_increment();
$(ASSERT get_s() ~= ?;)
   $(some sequence, whose histogram is to be formed, has
   been initialized)
DECLARATIONS
   seq n;
   INTEGER p, p1, p2, k;
   word w;
   truncated_word tw;
```

```
BEGIN
   n <- get_s();
   k <- get_p();
   EXECUTE w <- string(n, k+1)
      $(get the next word from the sequence)
      THEN
         ON no_word : RAISE(no_more_words);
         ON NORMAL : ;
      END;
   tw <- truncation(w); $(truncate the word)
   p <- hash(tw, plen); $(generate a hash address)
   FOR p1 FROM p TO p + plen - 1 UNTIL done DO
      $(search for appearance of w or an empty slot by
        attempting to insert)
      p2 <- p1 MOD plen;
      EXECUTE save(tw, p2) THEN
         ON hit : $(found tw; add count and finish)
               add_count(p2);
               SIGNAL(done);
         ON wrong_word : ; $(word is not tw; continue search)
         ON RESOURCE_ERROR : RAISE(RESOURCE_ERROR);
         ON NORMAL : SIGNAL(done);
            $(found an empty slot; save has inserted word; finish)
         END;
   THEN
      ON done :
            store_p(k + 1);
            RETURN;
      ON NORMAL : $(search for tw or empty slot failed; no more room)
         RAISE(RESOURCE_ERROR);
      END;
END;

OPROG reset_tally();
BEGIN
   reset_query();
   store_p(0);
END;


END_IMPLEMENTATION
```

Table 24: Query Implementation

```
IMPLEMENTATION query IN_TERMS_OF sequences, intarrays;


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };


    PARAMETERS

intarray director_array, count_array;
seq unique_sequence;


    EXTERNALREFS

    FROM query:
INTEGER plen;
VFUN get_string(INTEGER j) -> word w;
VFUN check_count(INTEGER j) -> INTEGER v;
VFUN size() -> INTEGER v;
OFUN save(word w; INTEGER p);
OFUN add_count(INTEGER p);
OFUN swap_query(seq n; INTEGER i, j);
OFUN reset_query();

    FROM sequences:
seq: DESIGNATOR;
VFUN string(seq n; INTEGER j) -> word w;
VFUN seqlen(seq n) -> INTEGER v;
OVFUN create_seq() -> seq n;
OFUN clear_seq(seq n);
OFUN append(seq n; word w);
OFUN swap_seq(INTEGER i, j);

    FROM intarrays:
intarray: DESIGNATOR;
INTEGER leni;
VFUN getint(intarray m; INTEGER j) -> INTEGER v;
OVFUN create_intarray() -> intarray m;
OFUN change_int(intarray m; INTEGER j, v);


    INITIALIZATION

DECLARATIONS
   INTEGER i;
BEGIN
   count_array <- create_intarray();
   director_array <- create_intarray();
   unique_sequence <- create_seq();
   FOR i FROM 1 TO leni DO
      change_int(director_array, i, 0);
      END;
   plen <- leni;
END;


    IMPLEMENTATIONS

VPROG get_string(INTEGER j) -> word w;
BEGIN
   EXECUTE w <- string(unique_sequence, j) THEN
      ON no_word : RAISE(no_word);
      ON NORMAL :
      END;
END;

VPROG check_count(INTEGER j) -> INTEGER v;
BEGIN
   IF j < 1 OR j > seqlen(unique_sequence)
      THEN RAISE(no_word);
      END_IF;
   v <- getint(count_array, j);
      $(no exception will occur since all intarrays are long enough
        to hold information about all words currently stored)
   RETURN;
END;

VPROG size() -> INTEGER v;
BEGIN
   v <- seqlen(unique_sequence);
   RETURN;
END;

OPROG save(word w; INTEGER p);
DECLARATIONS
   INTEGER j, j1;
BEGIN
   EXECUTE j <- getint(director_array, p+1) THEN
      ON no_int : RAISE(no_dir);
      ON NORMAL : ;
      END;
   IF j = 0 THEN
      $(slot is empty, try to append word with count of 1)
      EXECUTE append(unique_sequence, w) THEN
         ON RESOURCE_ERROR : RAISE(RESOURCE_ERROR);
         ON NORMAL :
            j1 <- seqlen(unique_sequence);
            change_int(director_array, p, j1);
            change_int(count_array, j1, 1);
               $(no exceptions can be returned from above calls
                  since it has been determined that there is an
                  empty slot in the director_array, which then
                  implies that there is more room in the
                  count_array)
               RETURN;
            END;
         END_IF;
      $(slot is not empty; determine if word there is w or another
      word)
      IF w = string(unique_sequence, j)
         THEN RAISE(hit);
         ELSE RAISE(wrong_word);
         END_IF;
   END;

   OPROG add_count(INTEGER p);
   DECLARATIONS
      INTEGER j;
   BEGIN
      EXECUTE j <- getint(director_array, p+1) THEN
         ON no_int : RAISE(no_dir);
         ON NORMAL: ;
         END;
      IF j = 0
         THEN RAISE(empty_slot);
         END_IF;
      change_int(count_array, j, getint(count_array, j) + 1);
   END;

   OPROG swap_query(INTEGER i, j);
   BEGIN
      EXECUTE swap_seq(unique_sequence, i, j) THEN
         ON no_word1 : RAISE(no_word1);
         ON no_word2 : RAISE(no_word2);
         ON NORMAL : ;
         END;
   END;

   OPROG reset_query();
   DECLARATIONS
      INTEGER i;
   BEGIN
      clear_seq(unique_sequence);
      FOR i FROM 1 TO leni
         DO change_int(director_array, i, 0);
         END;
   END;


   END_IMPLEMENTATION
```

Table 25: Sequences Implementation

```
IMPLEMENTATION sequences IN_TERMS_OF vcarrays, intarrays,
                                     vc_intarray_pairs;


    TYPES

word: { VECTOR_OF CHAR vc | LENGTH(vc) > 0 };
vc_intarray_pair:
   STRUCT_OF(vcarray vcarray_part; intarray intarray_part);


    PARAMETERS

intarray nstrings;


    EXTERNALREFS

    FROM sequences:
seq: DESIGNATOR;
VFUN string(seq n; INTEGER j) -> word w;
VFUN seqlen(seq n) -> INTEGER v;
OVFUN create_seq() -> seq n;
OFUN clear_seq(seq n);
OFUN append(seq n; word w);
OFUN swap_seq(seq n; INTEGER i, j);

    FROM vcarrays:
vcarray: DESIGNATOR;
VFUN char(vcarray vca; INTEGER j) -> CHAR c;
VFUN int_for_vcarray(vcarray vca) -> INTEGER v;
OFUN one_more_char(vcarray vca; CHAR c);
OFUN remove_chars(vcarray vca; INTEGER i);
OFUN clear_vcarray(vcarray vca);

    FROM intarrays:
intarray: DESIGNATOR;
INTEGER leni;
VFUN getint(intarray m; INTEGER j) -> INTEGER v;
OFUN change_int(intarray m; INTEGER j, v);

    FROM vc_intarray_pairs:
OVFUN create_vc_intarray_pair() -> vc_intarray_pair vnp;


    TYPE_MAPPINGS

seq : vc_intarray_pair;
```

```
    INITIALIZATION

DECLARATIONS
   INTEGER i;
BEGIN
   nstrings <- create_intarray();
END;


    IMPLEMENTATIONS

VPROG string(seq n; INTEGER j) -> word w;
DECLARATIONS
   vcarray vc;
   intarray m;
   INTEGER i, k, lower, upper;
BEGIN
   vc <- n.vcarray_part;
   i <- int_for_vcarray(vc);
   k <- getint(nstrings, i);
   IF j < 1 OR j > k
      THEN RAISE (no_word);
      ELSE
         m <- n.intarray_part;
         upper <- getint(m, 2*j);
         lower <- getint(m, 2*j-1);
         w <- VECTOR(FOR p FROM 0 TO upper - lower :
                          char(vc, p + lower));
         RETURN;
   END_IF;
END;

VPROG seqlen(seq n) -> INTEGER v;
BEGIN
   v <- getint(nstrings, int_for_vcarray(n.vcarray_part));
   RETURN;
END;

VPROG create_seq() -> seq n;
BEGIN
   EXECUTE n <- create_vc_intarray_pair() THEN
      ON RESOURCE_ERROR : RAISE(RESOURCE_ERROR);
      ON NORMAL :
          change_int(nstrings, int_for_vcarray(n.vcarray_part), 0);
          RETURN;
      END;
END;

OPROG clearseq(seq n);
BEGIN
      change_int(nstrings, int_for_vcarray(n.vcarray_part), 0);
END;
```

```
OPROG append(seq n; word w);
DECLARATIONS
   vcarray vc;
   intarray m;
   INTEGER i, j, k, q, r;
BEGIN
   vc <- n.vcarray_part;
   i <- int_for_vcarray(vc);
   j <- getint(nstrings, i);
   r <- 2 * j;
   $(the following statement determines if there is room in the
    intarray to store boundary positions for an additional word)
   IF r + 2 > leni
      THEN RAISE(RESOURCE_ERROR);
      END_IF;
   $(now put the characters of w into the vcarray one at a time)
   FOR k FROM 1 TO LENGTH(w) DO
      EXECUTE one_more_char(vc, w[k]) THEN
         ON RESOURCE_ERROR :
          $(no more room; remove inserted characters)
            remove_chars(vc, k-1);
            RAISE(RESOURCE_ERROR);
         ON NORMAL : ;
         END;
      END;
   $(set length, lower and upper boundaries)
   change_int(nstrings, i, j+1);
   m <- n.intarray_part;
   q <- getint(m, r);
   change_int(m, r+1, q+1);
   change_int(m, r, q+p);
END;

OPROG swap_seq(seq n; INTEGER i, j);
DECLARATIONS
   vcarray vc;
   intarray m;
   INTEGER p, q, r, s, t;
BEGIN
    $( check index bounds)
   vc <- n.vcarray_part;
   p <- int_for_vcarray(vc);
   q <- getint(nstrings, p);
   IF i < 1 OR i > q
      THEN RAISE(no_word1);
      END_IF;
   IF j < 1 OR j > q
      THEN RAISE(no_word2);
      END_IF;
    $( swap upper and lower bounds)
   m <- n.intarray_part;
```

```
    r <- 2 * i;
    s <- 2 * j;
    t <- getint(m, r);
    change_int(m, r, getint(m, s));
    change_int(m, s, t);
    r <- r - 1;
    s <- s - 1;
    t <- getint(m, r);
    change_int(m, r, getint(m, s));
    change_int(m, s, t);
END;


END_IMPLEMENTATION
```

Table 26: Seq_Pointer_Cells Implementation

```
IMPLEMENTATION seq_pointer_cells IN_TERMS_OF sequences, vcarrays,
                              intarrays, vc_etc_cells;


    TYPES

vc_intarray_pair:
  STRUCT_OF(vcarray vcarray_part; intarray intarray_part);


    EXTERNALREFS

    FROM seq_pointer_cells:
VFUN get_s() -> seq s;
VFUN get_i() -> INTEGER v;
OFUN store_s(seq s);
OFUN store_p(INTEGER i);

    FROM sequences:
seq: DESIGNATOR;

    FROM vcarrays:
vcarray: DESIGNATOR;

    FROM intarrays:
intarray: DESIGNATOR;

    FROM vc_etc_cells:
VFUN v_get() -> vcarray vc;
VFUN v_get_n() -> intarray m;
VFUN v_get_i() -> INTEGER v;
OFUN v_store(vcarray vc);
OFUN v_store_n(intarray m);
OFUN v_store_i(INTEGER i);


    TYPE_MAPPINGS

seq: vc_intarray_pair;


    IMPLEMENTATIONS

VPROG get_s() -> seq s;
BEGIN
   s <- STRUCT(v_get(), v_get_n());
   RETURN;
END;

VPROG get_p() -> INTEGER p;
```

```
BEGIN
   p <- v_get_i();
   RETURN;
END;

OPROG store_s(seq s);
BEGIN
   v_store(s.vcarray_part);
   v_store_n(s.intarray_part);
END;

OPROG store_p(INTEGER i);
BEGIN
   v_store_i(i);
END;


END_IMPLEMENTATION
```

Table 27: Vcarrays Implementation

```
IMPLEMENTATION vcarrays IN_TERMS_OF chararrays, intarrays;


    PARAMETERS

intarray length_array;


    EXTERNALREFS

    FROM vcarrays:
vcarray: DESIGNATOR;
INTEGER v_int_bounds;
VFUN char(vcarray vc; INTEGER i) -> CHAR c;
VFUN int_for_vcarray(vcarray vc) -> INTEGER i;
OVFUN create_vcarray() -> vcarray vc;
OFUN one_more_char(vcarray vc; CHAR c);
OFUN remove_chars(vcarray vc; INTEGER i);
OFUN clear_vcarray(vcarray vc);

    FROM chararrays:
chararray: DESIGNATOR;
VFUN getchar(chararray n; INTEGER j) -> CHAR c;
VFUN int_for_chararray(chararray n) -> INTEGER i;
OVFUN create_chararray() -> chararray n;
OFUN change_char(chararray n; INTEGER i; CHAR c);

    FROM intarrays:
intarray: DESIGNATOR;
VFUN getint(intarray m; INTEGER j) -> INTEGER v;
OVFUN create_intarray() -> intarray m;
OFUN change_int(intarray m; INTEGER j, v);


    TYPE_MAPPINGS

vcarray : chararray;


    INITIALIZATION

BEGIN
   length_array <- create_intarray();
   v_int_bounds <- leni;
END;


    IMPLEMENTATIONS

VPROG char(vcarray n; INTEGER i) -> CHAR c;
```

```
DECLARATIONS
   INTEGER j;
BEGIN
   j <- getint(length_array, int_for_chararray(n));
   IF i < 1 OR i > j
      THEN RAISE(no_char);
      ELSE c <- getchar(n, j);
           RETURN;
   END_IF;
END;

VPROG int_for_vcarray(vcarray n) -> INTEGER v;
BEGIN
   v <- int_for_chararray(n);
   RETURN;
END;

OVPROG create_vcarray() -> vcarray n;
DECLARATIONS
   INTEGER i;
BEGIN
   EXECUTE n <- create_chararray() THEN
      ON too_many_chararrays :
         RAISE(RESOURCE_ERROR);
      ON NORMAL:
         $(set length indicator to 0)
         i <- int_for_chararray(n);
         change_int(length_array, i, 0);
      END;
END;

OPROG one_more_char(vcarray n; CHAR c);
DECLARATIONS
   INTEGER i, j;
BEGIN
   j <- int_for_chararray(n);
   i <- getint(length_array, j);
   EXECUTE change_char(n, i+1, c) THEN
      ON no_char : RAISE(RESOURCE_ERROR);
      ON NORMAL :
         change_int(length_array, j, i+1);
      END;
END;

OPROG remove_chars(vcarray n; INTEGER i);
DECLARATIONS
   INTEGER j, k;
BEGIN
   j <- int_for_chararray(n);
   k <- getint(length_array, j);
   IF k < i
      THEN RAISE(underflow);
      ELSE change_int(length_array, j, k - i);
   END_IF;
END;

OPROG clear_vcarray(vcarray n);
DECLARATIONS
   INTEGER i;
BEGIN
   change_int(length_array, int_for_chararray(n), 0);
END;


END_IMPLEMENTATION
```

Table 28: Vc_Intarray_Pairs Implementation

```
IMPLEMENTATION vc_intarray_pairs IN_TERMS_OF vcarrays, intarrays,
                                 chararrays, chararrays_intarray_pairs;


    TYPES

vcarray_intarray_pair:
    STRUCT_OF(vcarray vcarray_part; intarray intarray_part);
chararray_intarray_pair:
    STRUCT_OF(chararray chararray_part; intarray intarray_part);


    EXTERNALREFS

    FROM vc_intarray_pairs:
OVFUN create_vc_intarray_pair() -> vc_intarray_pair vnp;

    FROM vcarrays:
vcarray: DESIGNATOR;
OVFUN create_vcarray() -> vcarray vc;

    FROM intarrays:
intarray: DESIGNATOR;
OVFUN create_intarray() -> intarray m;

    FROM chararrays_intarray_pairs:
OFUN store_chararray_intarray_pair(chararray_intarray_pair cnp);

    FROM chararrays:
chararray: DESIGNATOR;


    TYPE_MAPPINGS

vcarray: chararray;


    IMPLEMENTATIONS

OVPROG create_vc_intarray_pair() -> vc_intarray_pair vnp;
DECLARATIONS
   vcarray vc;
   intarray m;
BEGIN
   EXECUTE vc <- create_vcarray() THEN
      ON RESOURCE_ERROR : RAISE(RESOURCE_ERROR);
      ON NORMAL : ;
      END;
   EXECUTE m <- create_intarray() THEN
      ON RESOURCE_ERROR : RAISE(RESOURCE_ERROR);
```

```
        ON NORMAL : vnp <- STRUCT(vc, m);
                    store_chararray_intarray_pair(vnp);
        END;
    END;

    END_IMPLEMENTATION
```

Table 29: Vc_Etc_Cells Implementation

```
IMPLEMENTATION vc_etc_cells IN_TERMS_OF vcarrays, intarrays,
                                 chararrays_etc_cells;


    EXTERNALREFS

    FROM vc_etc_cells:
VFUN v_get() -> vcarray vc;
VFUN v_get_n() -> intarray m;
VFUN v_get_i() -> INTEGER i;
OFUN v_store(vcarray vc);
OFUN v_store_n(intarray m);
OFUN v_store_i(INTEGER i);

    FROM vcarrays:
vcarray: DESIGNATOR;

    FROM intarrays:
intarray: DESIGNATOR;

    FROM chararrays_etc_cells:
VFUN c_get() -> chararray n;
VFUN c_get_n() -> intarray m;
VFUN c_get_i() -> INTEGER v;
OFUN c_store(chararray n);
OFUN c_store_n(intarray m);
OFUN c_store_i(INTEGER i);


    TYPE_MAPPINGS

vcarray: chararray;


    IMPLEMENTATIONS

VPROG v_get() -> vcarray vc;
BEGIN
   vc <- c_get();
   RETURN;
END;

VPROG v_get_n() -> intarray m;
BEGIN
   m <- c_get_n();
   RETURN;
END;

VPROG v_get_i() -> INTEGER i;
BEGIN
```

```
   i <- c_get_i();
   RETURN;
END;

OPROG v_store(vcarray vc);
BEGIN
   c_store(vc);
END;

OPROG v_store_n(intarray m);
BEGIN
   c_store_n(m);
END;

OPROG v_store_i(INTEGER j);
BEGIN
   c_store_i(j);
END;


END_IMPLEMENTATION
```