THE UNITS OF MEASURE CONSISTENCY CHECKER FOR THE ENTITY-RELATIONSHIP-ATTRIBUTE REQUIREMENTS MODEL

by

GALE LYNN METZ

B. S., University of Illinois, Urbana, 1977

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

Kansas State University
Manhattan, Kansas

1986

Approved by:

Major Professor

Table of Contents

Chapter One

Introduction

1.1 Overview

This report describes a units of measure consistency checker. The consistency checker has been developed as part of a group implementation project. The group project is intended to provide a prototype of a software development environment. This portion of the system will verify that the output units of measure specified in the prototype's software requirements specification can be derived from the given input units of measure.

The report is organized into five chapters. The first chapter continues with a literature review. The literature review demonstrates a need for a more formalized and automated software development environment. The chapter ends with a summary of the units of measure consistency checker in relation to the literature review.

The second chapter presents a discussion of the requirements necessary for the consistency checker to be a useful tool within the prototype environment. The chapter begins with a description of the Entity - Relationship - Attribute (E-R-A) Requirements Specification. This specification is the input used to drive the consistency checker. The chapter continues with a discussion of the error and warning conditions flagged by the consistency checker. Finally, the chapter concludes with a discussion of the error processing and the operating environment of the units of measure consistency checker.

The next chapter presents the design of the consistency checker. It begins with an overview of the high level design, including a hierarchy diagram of the consistency checker's structure. The chapter continues with a detailed description of the modules in the consistency checker. These descriptions include the major data structures used by each of these modules.

The fourth chapter discusses the implementation of the consistency checker. It addresses the size of the various modules and the details of the testing used to insure their

correct operation. The chapter also includes the integration testing used to insure the correct operation of the units of measure consistency checker.

The final chapter presents the conclusions and extensions. These extensions include design changes as well as increased capabilities.

The report is concluded with several attachments. They are a BNF description of the E-R-A Requirements Specification language; a sample E-R-A Specification; the function frame data structure; the i/o data frame data structure; the resolved relations frame data structure; a sample FUNCT data file; a sample IODATA data file; a sample OTDATA data file; a sample TDATA data file; a sample UDATA data file; sample long output data; sample short output data; and finally the code.

1.2 Large System Development

The movement today is toward larger, more complex software systems. These larger, more complex systems are due to computers gaining an ever increasing share of the work that has traditionally been performed manually. In addition to this traditional work, existing facilities and systems are being integrated with new facilities and systems to perform tasks that have never been possible before.

A small project can be defined as one in which a single individual can encompass and resolve all of the significant issues involved in developing the system [7]. Unlike these systems, large systems involve the efforts of many people. Even though these large systems are typically divided into smaller, more manageable pieces, they still present a number of problems and challenges not usually found in small projects.

Large systems are typified by a long development interval in which the mass of information involved quickly exhausts human capabilities to retain a clear picture of the system as a whole [21]. In this environment individual tasks are designed and programmed independently of other tasks. This inability to view the system as a whole, introduces problems in the area of communication and coordination in addition to common

programming problems. Therefore, help is needed in:

1) recording what is known and what has been decided about the system.

2) uncovering what is unknown.

3) assessing the suitability and completeness of the eventual system.

4) coordinating and monitoring the efforts of the team. [17]

These problems indicate that large system development must become a sequence of steps, starting with a formal requirements specification document. This sequence of steps is commonly called the software life cycle. It consists of several phases:

1) requirements

2) development

3) testing

4) user acceptance

5) operations and maintenance [6,16]

During the requirements phase, a large software system is taken from an abstract concept or proposal to a well defined software system. This system is described in the requirements specification document. The completed document can then be used as a contract between the client and the software development team. Therefore, due to its impact on the entire development cycle, the requirements phase is becoming an area of increased importance and is gaining more and more attention.

Unfortunately the poor quality of real-world requirements specification documents has seriously reduced their usefulness. Many of the errors in these documents are not found until late in the development cycle when they can seriously affect already completed and verified software. In the worst case they may necessitate the redesign and

recoding of a significant number of modules, thus negating any previously completed tests. Savings of up to 100:1 have been noted by finding and correcting these errors as early as possible in the life cycle [6]. Therefore a correct requirements specification should be created prior to beginning any actual software development. This step will ease many of the coordination and communication problems that can develop in a large system and it can be a major time and cost savings step in the development of a large software system.

The determination of the correctness of a requirements specification can be divided into two areas. The first area is validation. Validation is the process of determining whether the product fulfills the expectations of the user. The second area is verification. Verification is the process of determining the internal accuracy of the document itself. [6]

There are four basic criteria for determining the correctness of the requirements specification document in these areas. They are completeness, consistency, feasibility, and testability. Completeness establishes that each part is present and fully developed. Consistency determines that the provisions of the document do not conflict with each other. Feasibility requires that the benefits of the specified system exceed the cost of its life cycle. The final criteria, Testability requires that there is an economically feasible technique for determining whether or not the developed software will satisfy the requirements specification document.

Traditionally, verification of the requirements specification has been a manual process, due to the use of informal English in these documents. This process can be costly and time consuming since it may involve the cooperation of several people in different areas. Therefore, in order to automate some of this verification process, there has been an increased interest in formalizing the grammar and structure of the requirements specification document. Several different models have been developed. They are briefly described in the following sections.

1.2.1 Finite-State Machines

The Finite-State Machine model describes which outputs occur in response to

the last input received and the machine's current internal state. This model is difficult to use during the requirements phase since the system states have not yet been defined. However, it can be modified to use user function states. This model can become quite complex, especially in the case of the augmented-state-transition matrix used for synchronizing real time systems. Therefore, because of this complexity, the model can be very difficult for a non-technical client to understand[10]. SADT, developed by Softech, Inc., uses a type of Finite - State Machine with user function states approach[26][27].

1.2.2 Stimulus-Response Sequences

The Stimulus-Response sequence model describes the system responses that result from the user stimuli in an algorithmic English-like notation. The states in this model correspond to the states in the finite-state machine model. The key advantage to this model is that the description can be easily read and understood by a non-technical client. In addition, the descriptions can be easily modularized[10].

1.2.3 General Bi-Partite Graphs

A Bi-Partite graph divides its nodes into two disjoint sets. These sets are usually represented by different geometrically shaped nodes (ex. circle and diamond). The nodes in these sets are then connected by arcs. These arcs may not connect two nodes from the same set. One way of using the model to define requirements would be to assign function states to one set and system status to the other set. The arcs can then be labeled with stimuli[10].

1.2.4 Petri Nets

Petri Nets are typically used to define synchronization among parallel processes. They are a type of bi-partite graph, which use circles to represent states and bars to represent synchronization points. Circles with no entering arcs are used to indicate stimuli. Circles with entering arcs are considered function states, with the arcs referring to system responses. The bars are then used to refer to points where synchronization checks must be made. This model is also not suited to a non-technical client. However, they are

highly useful to the designer[10].

1.2.5 Input-Process-Output Sequences

The Input-Process-Output sequence model is typically used for transaction oriented requirements. It emphasizes what data is translated into what data by what process. This model describes the system as a disjoint set of processes with inputs and outputs[10]. PSL/PSA, developed at the University of Michigan, uses a type of Input - Process - Output Sequences approach[20].

1.2.6 Requirements Nets

A Requirements Net defines the sequence of processes to be performed in response to an external stimulus. Each requirements net corresponds to one state of a user function. The net describes how the system should respond given the stimulus and current state of the system. This description is given in graphical form[10]. SREM, developed by TRW, uses an artificial language (RSL) to define requirements nets type approach[25].

1.3 Summary of Project

As software systems become larger and more complex, many man hours can be wasted designing and testing software that has been developed from incorrect specifications. Therefore, increasing effort is being devoted to uncovering errors as early as possible in the development cycle. Much of this effort is being directed at formalizing the language used in the requirements specification document. This formalization will eventually allow computer aids to help ensure the correctness of the requirements specification document.

One such formalism is the Entity - Relationship - Attribute Requirements Specification language. It combines the Input - Process - Output sequence model, discussed previously, with the Entity - Relationship model, developed by Peter Pin-Shan Chen to describe data[9]. This language uses a frame oriented structure to describe a system as a set

of entities and a set of relations/attributes among those entities. According to Chen, an entity can be defined as a "thing" which can be distinctly identified. He defines a relationship as an association among these entities and an attribute as a function which maps from an entity set or a relationship set into a value set[9]. See Appendix A for the BNF syntax description and Appendix B for a sample specification.

This type of representation allows the specification to be machine checked for various forms of correctness. One form of correctness verification that can be automated is the verification of the consistency of the units of measure use in the requirements specification document. Units of measure can be defined as a means of referring to a quantity of some quality, activity, or substance[1]. By verifying that the specified output units of measure can be obtained from the given input units of measure and the units of measure attached to any constants the function has access to, one can assume that each function has access to all necessary data.

Chapter Two

Requirements

2.1 Overview

The definition of requirements for the units of measure consistency checker is divided into three sections. The first section describes the input the units of measure consistency checker is expected to process. The second section describes the output the consistency checker is expected to produce in response to the given input, and the final chapter describes the environment the consistency checker is expected to operate in.

2.2 Input

The units of measure consistency checker must be able to use the E-R-A Requirements Specification document text as input (See Appendix A for the BNF syntax description of the E-R-A Requirements Specification language). The specification consists of the specification name and the specification body, followed by a mode table. The units of measure consistency checker is only required to process the body of the E-R-A Requirements Specification document.

The body of the requirements specification document consists of a set of frames. Each frame contains information describing a given entity. The entity types currently defined in the E-R-A Specification language are:

Activity

Periodic_function

Input - defines input relations

Output - defines output relations

Input_output - defines relations used as both input and output

Type - defines structure and type relations

Data - defines uses relations

Comment - defines a comment

Within the BNF syntax description for the E-R-A Requirements Specification, entity types have been divided into two categories. These categories have been named function entity types and i/o data entity types (See the BNF definition in Appendix A). The i/o data entity definitions further define the information provided in function definitions or other i/o data definitions. The function entity types are Activity and Periodic_function. The other entity types are i/o data entity types.

The body of each entity frame definition usually consists of information in the form of relations between a given entity and other entities within the specification. In addition, information may also be provided in the form of attributes, which provide information about a given entity without referring to any other entities. Each relation/attribute is specified by a keyword. The relation/attribute keywords currently defined in the E-R-A Specification language are:

keywords

input - defines input to a function entity

output - defines output from a function entity

required_mode

necessary_condition

occurrence

assertion

action

comment - defines a comment within an entity frame

media

structure - defines a structure

type - defines a data type

units - defines the units associated with a data item

subpart_is

subpart_of

uses - defines data a function entity

has access to

Each relation/attribute keyword in the specification has a value associated with it. The value of a relation is either the name of the i/o data entity defining that relation, or a combination of text and the i/o data entity name. The value of an attribute is a text description of that value.

Each relation/attribute definition starts on a new line. If a relation/attribute definition continues onto another line, the continuation line starts with a blank keyword field followed by a colon. These continuations are interpreted as an "and" condition in the relation/attribute definition. Multiple occurrences of a relation/attribute keyword represent an "or" condition within the entity definition. For example:

Activity : name1

input : $name2$

: $name3$

input : $name4$

The occurrence of $name2$ and $name3$ should be interpreted as meaning both inputs $name2$ and $name3$ are required if that input relation is being used. The occurrence of another relation definition $name4$ should be interpreted as an "or" condition. This "or" condition means that the activity is either expecting $name2$ and $name3$ as input or $name4$. See Appendix B for a sample E-R-A Requirements Specification.

The E-R-A Requirements Specification document will be entered into the units of measure consistency checker in the form of a text file. The consistency checker will attempt to open the file and read it as input. The input requirements specification in this file will have several constraints:

1) It must be syntactically correct. This assumption is

reasonable since the E-R-A Requirements Specification

document can be assumed to be machine generated.

2) This input document must also be consistent and complete in terms of the entity definitions. The units of measure consistency checker is not required to identify any of these errors, since it is intended as part of a larger, more complete verification system.

2.3 Output

The output requirements definition is divided into two sections. The first section describes the output of the units of measure consistency checker during normal operation. The final section describes the conditions that produce error and warning messages not associated with the normal operation of the consistency checker.

2.3.1 Normal Processing

The main objective of the units of measure consistency checker is to verify that the specified output units of measure can be obtained from the given input units of measure and any units of measure the function entity has access to through the uses relations, or the synonym and conversion definitions. Currently there are two function entity types which must be processed by the units of measure consistency checker. They are the Activity and the Periodic_function entity types.

In order to determine the units of measure associated with the input, output and uses relations defined in the function entity frames, the units of measure consistency checker must use the i/o data entity definitions. The i/o data entity types used to resolve the input relations are the Input and Input_output entity types. There are also two entity types used to resolve the output relations. They are the Output and Input_output entity types. The uses relations can be resolved by using the Constant and Data i/o data entity types.

Within the i/o data entity frames, the units of measure consistency checker must recognize three types of relations. They are the structure, type and units relations. If any of the i/o data entity frames used to resolve the input, output and uses relations, do not contain units attribute definitions, the units of measure consistency checker should attempt to resolve any type or structure relations found in the i/o data frames using the Type i/o data entity definitions.

After all relations associated with the input, output and uses definitions expressed in the function entity frames have been resolved, the units of measure consistency checker should verify that the output units of measure can be obtained from the input units of measure and the uses units of measure. In order to verify the units of measure within the function definitions, the units of measure consistency checker must interpret the input, output and uses definitions as a set of "and" and "or" conditions. For example, the input relations within a function definition may resolve into unit1 and unit2 and unit3 or unit4 and unit5. These "and" and "or" conditions can be determined from the structure of the entity frame described in the Input section of this chapter. In addition, "and" and "or" conditions can be defined by including the text "and" and/or "or" within the value of the relation.

Each input and-set will be compared to each output and-set. Any units in the output and-set which have not been found in the input and-set should be searched for in the uses set ("And" conditions and "or" conditions are not distinguished in the uses set). If any units in the output and-set have not been found, a warning message will be printed. If the output and-set cannot be produced by any input and-set and the uses set, an error message will be printed.

The matching function should be as strict or as liberal as the user determines is necessary. Therefore, two additional attempts should be made to match the output and-set. The first additional attempt should check a set of acceptable synonyms for the unmatched unit. The second method should attempt to match the unit using an acceptable

conversion for the unit. If the unit is matched through using a conversion, a conversion warning message should be printed.

The synonym and conversion sets should be user defined. If no synonyms or conversions are provided, the matching function is very strict. The user may also provide only synonyms or only conversions, or the user may choose to provide both synonyms and conversions. This capability allows the matching function a large degree of versatility.

The user should also be able to specify either a short or a long report type. The long report type will print out the function frames as the units of measure consistency checker processes them, along with any errors or warnings it may encounter. The short report type will only print out function frames containing errors and/or warnings, and the associated errors and/or warnings.

2.3.2 Error Processing

Several types of error and warning conditions may occur during the normal processing of the units of measure consistency checker. These conditions fall into three categories. They are the operating environment, physical limitations within the program, and the input specification. The units of measure consistency checker will attempt to recover from any problems it encounters. If it is unable to recover, it will print an error message and terminate. Otherwise it will print a warning message and continue. These conditions and messages will be discussed in detail in the design portion of this report.

2.4 Operating Environment

The units of measure consistency checker is intended as part of a much larger system. Therefore, its interface should be as simple as possible. It should also be written in C - Language and run on a UNIX operating system.

Chapter Three

Design

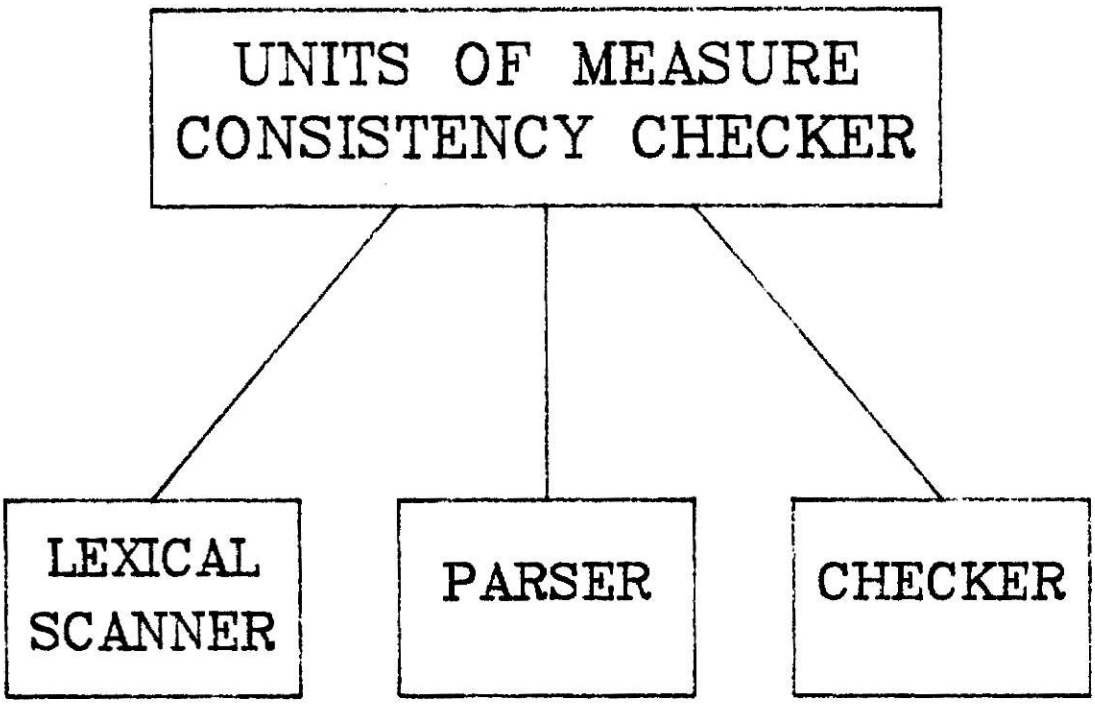
3.1 Overview

The design chapter is divided into two sections. The first section present the high level design of the units of measure consistency checker. The second section presents the detailed design of the consistency checker, including descriptions of each module and the data structures used within it.

3.2 High Level Design

The units of measure consistency checker can be functionally divided onto three parts. They are the lexical scanner, the parser and the checker (See Figure 1 for the High Level Structure Chart). The controlling program first calls the lexical scanner. The lexical scanner removes extraneous information from the input specification and places the remaining text into a .temp file. The controlling program then passes this .temp file to the parser.

The parser reads the .temp file as input. It then parses this file and from the information it finds in the text, it creates five data files. These files are named FUNCT, INDATA, OTDATA, UDATA, and TDATA. The FUNCT data file is built from the information found in the function frame definitions (See Appendix C.1 for the function frame data structure). The other data files, INDATA, OTDATA, UDATA, and TDATA, have the same format and are built from the information found in the i/o data frame definitions (See Appendix C.2 for the i/o data frame data structure). The information found in the Input and Input_output entity frame definitions is used to build the INDATA file. The OTDATA file is built using the Output and the Input_output entity frames. The Constant and Data entity frames are used to build the UDATA file and the Type entity frames are used to build the TDATA file (See Appendix D for an example of each data file). After the parser has completed its processing, the controlling program calls the checker.



HIGH LEVEL DESIGN

FIGURE 1

The checker uses the i/o data files to resolve the relations stored in the function frame data file. The frames from the data file are processed one frame at a time. The input relations, which are defined in the frame, are resolved by the checker using the information, which is stored in the INDATA file. Similarly, the OTDATA file is used to resolve the output relations and the UDATA file is used to resolve the uses relations. The TDATA file is then used to resolve any structure and/or type relations until the units definitions are found. If no relations remain to be resolved and no units have been found, the checker assumes that there are no units associated with that item.

After all the relations have been resolved, the checker attempts to verify that the specified output units of measure can be obtained from the given input units of measure and any units of measure the function has access to through the uses relations. In addition to the units of measure provided in the specification text the checker will attempt to verify the output units of measure using user defined synonyms and conversions. As the checker analyzes the units of measure, it prints the units the units of measure being analyzed to a report file. After the units analysis for that function frame has been completed, the file is printed on the screen if the long report type has been requested. If the short type has been requested, the checker will only print the file on the screen if it contains error or warning messages.

3.3 Detailed Design

3.3.1 Controlling Program

The controlling program is a shell script which processes the command line and calls the scanner, the parser, and the checker. It first determines the report type and the input filename. It then calls the scanner using the input file as data and traps the output from the scanner in a temporary file. The name of this file is created by concatenating the input filename with ".temp". Next the controlling program calls the parser using the .temp

file as input. After the parser has completed processing the .temp file, the controlling program deletes the temporary file and calls the checker with the desired report type. After the checker has completed processing, the controlling program will determine if any other filenames were included on the command line. If any other filenames were included on the command line, it will process them as described above. When no files are left to process, the controlling program will terminate.

3.3.2 The Scanner

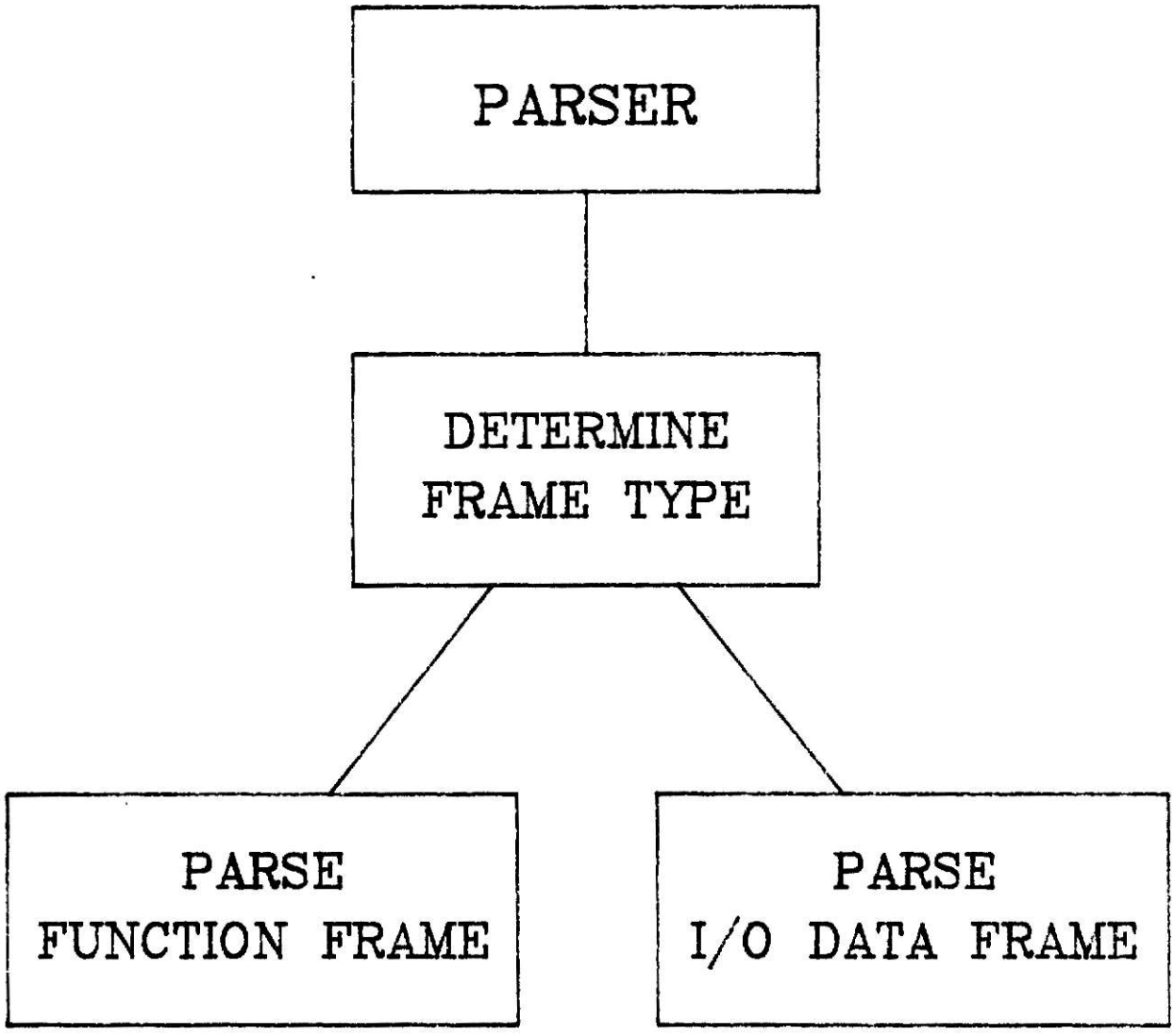
The scanner is an AWK program which prepares the input requirements specification for the parser. It first modifies each continuation line to include the proper keyword and replaces the colon with a "+" to indicate that it is a continuation line. It then eliminates any unnecessary entity and relation/attribute lines from the specification.

3.3.3 The Parser

See Figure 2 for the Parser Structure Chart.

3.3.3.1 Main Program

The parser is a C-Language program which accepts a filename as input. The main routine attempts to open the input file. If it cannot open the input file, it prints an error message and terminates. Otherwise, it calls the routine, determine frame type. When determine frame type has completed, the routine closes the input file and terminates.



PARSER DETAIL DESIGN

FIGURE 2

3.3.3.2 Determine Frame Type

The main purpose of this routine is to build the files FUNCT, INDATA, OTDATA, UDATA and TDATA. It first creates the five output files. If it is unable to create any of these files, the routine will print an error message and terminate. If the files were created successfully, the routine searches for the start of an entity frame. If any lines containing text are flushed while the routine is searching for the start of a frame, a warning message and the flushed line will be printed.

Next the routine determines the entity type. If the entity type is either Activity or Periodic_function, the routine calls the parse function routine. The frame structure returned by the parse function routine is then stored in the FUNCT file. The i/o data frames are handled in a similar manner. The routine which creates the i/o data frame structure is called parse i/o data. The i/o data frame structures are then stored in the appropriate file. If the determine frame type routine encounters any entity types other that Activity, Periodic_function, Input, Output, Input_output, Constant, Data, or Type, a warning message and the flushed lines will be printed. After all the frames in the input file have been processed, the routine closes the files it has created and terminates.

3.3.3.2.1 Parse Function Frame

The main purpose of the parse function routine is to build the function frame data structure (See Appendix C.1 for the data structure format and see Appendix D.1 for parsed function frame examples). This routine accepts the current line, the input file pointer, and a pointer to the function data structure as input. The function data structure is populated by processing lines from the input file one at a time until a frame separator is found. The routine first parses the frame header and stores the resulting information in the function frame data structure. After the header has been parsed and stored, the routine processes the body of the frame.

Within the body of the frame, the routine attempts to locate input, output, and

uses relations. If any other relation types are encountered a warning message will be printed and the line will be flushed. Even though the scanner eliminates unnecessary relations from the input specification, the BNF allows structure, type, and units relations/attributes in function frame definitions also. Therefore, since the units of measure consistency checker is not required to process these relations in the context of a function frame definition, they will be eliminated here.

The routine uses the values of the input, output, and uses relations to populate the function data structure. The input, output, and uses structures, in the function frame data structure, are two dimensional arrays. The first dimension represents the "and" conditions found within the function frame and the second dimension represents the "or" conditions encountered within the frame. If either array limit is reached, while these structures are being populated, a warning message will be printed and any other relation/attribute values for that position will be flushed. In addition, if the routine encounters any line which cannot be parsed, the line will be flushed and a warning message will be printed.

3.3.3.2.2 Parse I/O Data Frame

The main purpose of the parse i/o data routine is to build the i/o data frame data structure (See Appendix C.2 for the structure format and Appendix D for parsed i/o data frame examples). This routine accepts the current line, the input file pointer, and the pointer to the i/o data structure as input. The i/o data structure is populated by processing lines from the input file one at a time until the frame separator is found. The routine first parses the frame header and then stores the resulting information in the i/o data frame data structure.

Within the body of the frame, the routine attempts to locate type, structure, and units relation/attribute definitions. If any other relation types are found, a warning message will be printed and the line will be flushed. Even though the scanner eliminates unnecessary relations from the input specification, the BNF allows input, output, and uses

relations/attributes in i/o data frame definitions. Therefore, since the units of measure consistency checker is not required to process these relations in the context of an i/o data frame definition, they will be eliminated here.

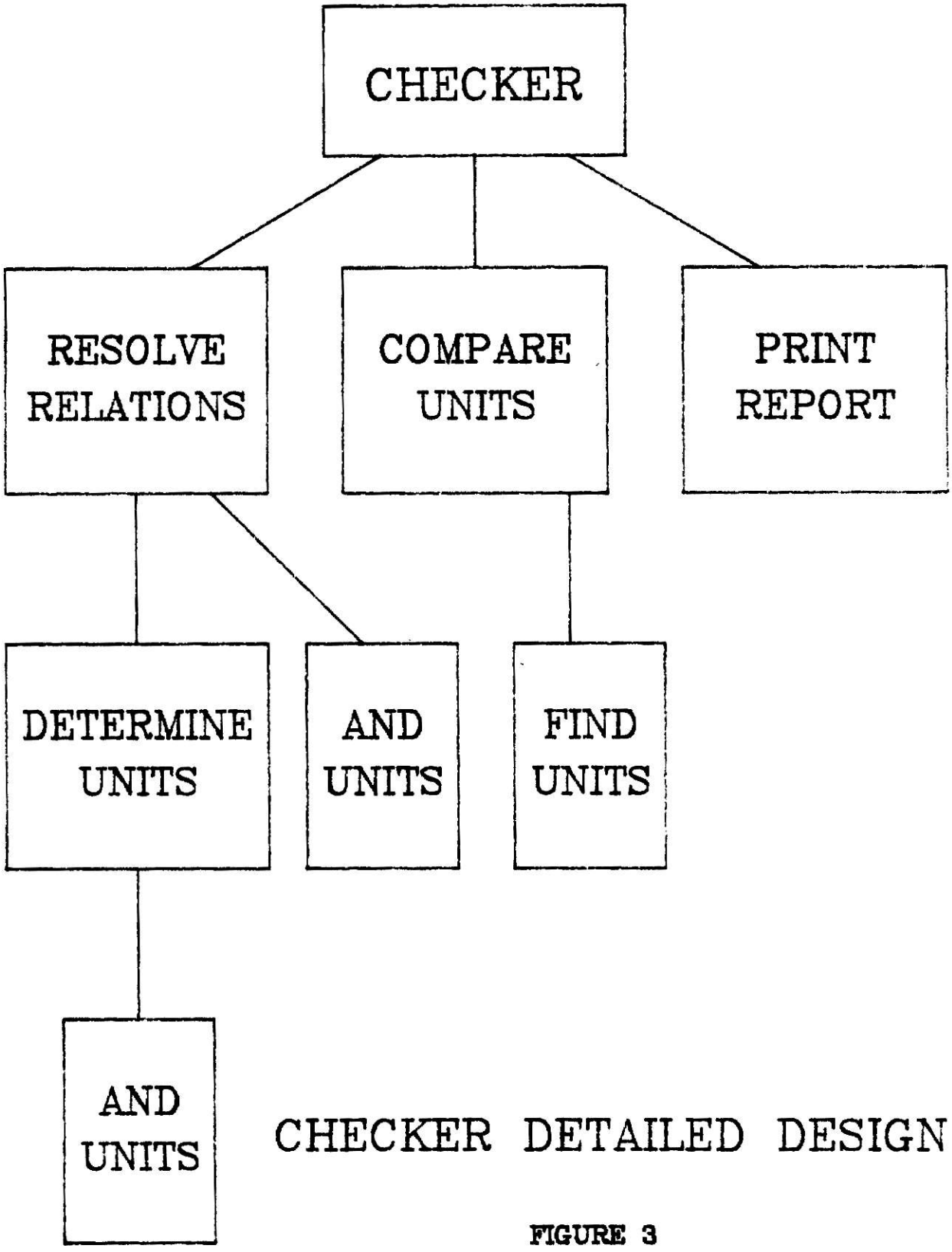
The routine uses the values of the type, structure, and units relations/attributes to populate the i/o data data structure. The type, structure, and units structures, in the i/o data frame structure, are two dimensional arrays. The first dimension represents the "and" conditions found within the i/o data frame and the second dimension represents the "or" conditions encountered within the frame. If either array limit is reached, while these structures are being populated, a warning message will be printed and any other relation/attribute values for that position will be flushed. In addition, if the routine encounters any lines which cannot be parsed, the line will be flushed and a warning message will be printed.

3.3.4 The Checker

See Figure 3 for the Checker Structure Chart.

3.3.4.1 Main Program

The checker is a C-Language program which accepts the report type as input. The main routine attempts to open the FUNCT file. If it cannot open the FUNCT file it prints an error message and terminates. Otherwise, it reads a function frame from the file and processes the frame by calling the resolve relations routine. The resolve relations routine will then build the resolved relations function frame data structure (See Appendix C.3 for the structure format). This structure is then passed to the compare units routine. After the compare units routine has completed, the main program calls the print error routine with the desired report type. The main routine continues to read frames and process them until an end of file is reached. The routine then closes the FUNCT file and deletes the FUNCT, INDATA, OTDATA, UDATA, and TDATA files.



CHECKER DETAILED DESIGN

FIGURE 3

3.3.4.2 Resolve Relations

The main goal of the resolve relations routine is to build the resolved relations data structure. This structure represents the input function frame with all of the units resolved. The routine first opens all of the i/o data frame files. If any of these files cannot be opened, an error message is printed to the screen and the routine is terminated.

In order to build the resolve relations frame data structure, the routine calls determine units with each relation in the function frame. If the relation used as input to the determine units routine is part of an "and" condition, the units returned by the routine are anded with the and-set in the resolved relations data structure. After the resolved relations data structure has been completed, the resolved relations routine closes all of the i/o data files and terminates.

3.3.4.2.1 Determine Units

The purpose of the determine units routine is to find the units associated with a given relation value. It is a recursive routine, which accepts a relation value and two file descriptors as input. The first file descriptor is a primary search file and the second file is a secondary search file. The primary search file is one of the INDATA, OTDATA, UDATA, or TDATA files and the secondary search file is always the TDATA file. The routine returns a pointer to a link list of and-sets. Each node within the link list represents an and-set and each link represents an "or" condition.

Determine units reads frames from the primary search file one at a time until a match for the given input relation is found. If a match is found, the routine will check the matching frame for a units definition. If a units definition is found, it is copied to the link list and the pointer to the link list is returned. If no match is found, "no match" is copied to the node and the pointer to the node is returned.

If no units definition is found in the matching frame, the frame will be searched for a type definition. If a type definition is found, determine units will call itself with the

secondary search file descriptor in both the secondary and the primary search file descriptor positions, for each relation in the type definition. If no type definition is found, the frame will be searched for a structure definition. If a structure definition is found, determine units will call itself with the secondary file descriptor in both the primary and the secondary search file descriptor positions, for each relation in the structure definition. If no structure definition is found, "no units" is copied to the node and the pointer to the node is returned.

If at any time the routine encounters any problems copying the units into the units structure, a warning message will be printed. The routine will also print a warning message if any value cannot be added to an and-set because the limit of the array has been reached. In addition, a warning message will be printed if the routine cannot allocate storage for a node.

3.3.4.2.2 And Units

And units is a utility routine, which is used to and two and-sets together. An and-set is a link list of arrays. Each element in the array represents an "and" condition and each node represents an "or condition. The routine first counts the number of nodes in the second input and-set. It then makes that number of copies of each node in the first input and-set. After all of the copies have been made, the routine copies each node in the second and-set into each node in the first and-set, and the routine is terminated.

If the routine encounters any problems while building the link list, a warning message will be printed. The routine will also print a warning message if any value cannot be added to an and-set because the limit of the array has been reached. In addition, a warning message will be printed if the routine cannot allocate storage for a node.

3.3.4.3 Compare Units

The purpose of the compare units routine is to create the error report file. It ac-

cepts the report type, the function frame, and the resolved relations frame as input. The routine first prints the function frame type and name to the file. After the frame type and name are printed to the file, the original relation values for the first and-set in the output definition are also printed to the file. Then, since any relation may resolve into any number of "and" and "or" conditions, the routine will process the units and-sets for the current set of output relations one at a time. After the first output units and-set is printed to the file, the routine will print the original relation values for the first and-set in the input definition, and the first input units and-set to the file.

After this information has been printed to the report file, the routine will attempt to locate each of the output units in the current input units and-set, using the utility function find unit. If any output units are not found in the current input set, the compare units routine will attempt to locate them any where in the uses definition. If any output units still remain unmatched, the routine will attempt to find them in the input set and the uses definition using user supplied synonyms for the missing units. If any units still remain unmatched, the compare units routine will attempt to locate them in the input set and the uses definition using user supplied conversions. If a match is found using a conversion, a conversion warning message will be printed to the report file. If there are any units remaining unmatched after all of the previously specified comparisons have been made, an unmatched units warning message will be printed to the report file.

The compare units routine will then compare this output and-set, in the manner described above, to each input and-set in the input relation set, while printing each input and-set to the error file. It will then print the next input relation set to the report file, and process all of its input and-sets as described above. This procedure will be repeated until all of the input and-sets have been compared. If no input and-set together with the uses definition is able to produced the entire output and-set, an error message will be printed to the report file. The compare unit routine will then process each output units and-set in the output relation as it processed the first output units and-set. After the first

output relation has been completed, the compare units routine will process the next output relations set. This procedure will be repeated until all of the output relation sets have been processed. After this processing has been completed, the routine will terminate.

3.3.4.3.1 Find Unit

Find unit is a utility which searches a given and-set for a given unit value. It accepts the unit value and a pointer to the and-set as input. It compares each unit value in the and-set to the given input unit value, until a match is found or no unit values in the and-set remain to be compared. If a match is found the utility returns "1". Otherwise, it returns "0".

3.3.4.4 Print Errors

The purpose of the print errors routine is to provide the desired report output. It accepts the desired report type as input. If the report type is short, it will first check the file for warning or error messages. If no error or warning messages are found, it will delete the file and terminate. Otherwise, it will print the contents of the file to the screen before it deletes the file and terminates. If the long report type is desired, it will simply print the file to the screen, delete the file and terminate.

Chapter Four

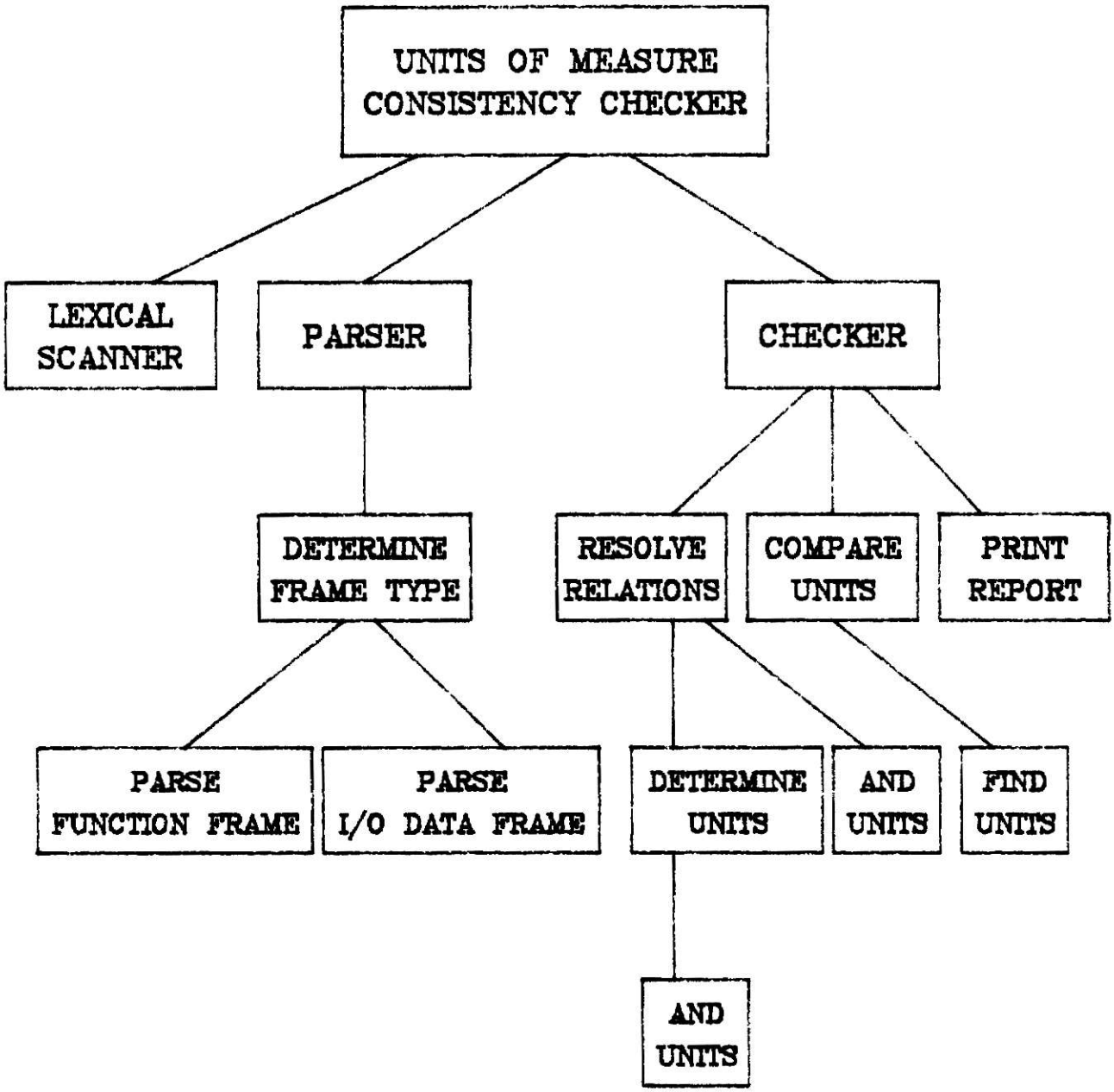
Implementation

Each of the major functional tasks in the units of measure consistency checker was developed as an independent program. The scanner is an awk program, which creates the input file for the parser. The parser is an independent C-Language program, which creates five output files. These files are then used by the checker, which is also an independent C-Language program. The checker then prints the final analysis to the screen (See Figure 4 for a complete hierarchy chart).

The scanner was developed first. It works in two phases. The first phase duplicates the appropriate keyword on any continuation line and replaces the colon with a "+" sign. The second phase eliminates any undesirable entities and relations/attributes from the text file. It is 16 lines of awk code.

Once the scanner was completed, the controlling shell script was developed. It is invoked by entering "umecc" and the input filenames. If the long report type is desired, "umecc" may be followed by the "-l" option before the filenames. Otherwise, the short report type is assumed. The controlling program was developed using dummy routines to stub out the parser and the checker. The controlling program is approximately 52 lines of shell script. It was verified using various forms of the command line and some echo statements.

The parser was developed in a top down manner. It is approximately 600 lines of code. First the main routine was developed. It is very simple and does little more than open the input file and call the determine frame type routine. Next the determine frame type routine was developed and tested using the data prepared by the scanner. Once the program was able to differentiate entity types correctly, the frame parsing routines were developed.



UNITS OF MEASURE CONSISTENCY CHECKER

STRUCTURE CHART

FIGURE 4

The first frame parsing routine to be developed was the parse function routine. After the parse function routine was tested and determined to be operating correctly, the parse i/o data routine was written and tested. Both the parse function and the parse i/o data routines were tested extensively using print statements.

After both routines appeared to be working properly, an independent print routine was written. This routine is not part of the units of measure consistency checker parser. It was approximately 150 lines of code and was written to open the five output files one at a time and print their contents to the screen. This final test verified the functioning of the parser.

Once the parser was completely developed and tested, the checker was begun. The checker, like the parser, was developed in a top down manner. It is approximately 900 lines of code. First, the main routine was developed. It reads a frame from the function frame file and calls the routines, resolve relations, compare units, and print errors, to process the frame. Once the main procedure was completed, the resolve relations routine was developed.

The resolve relations routine is a relatively simple routine, which calls the determine units routine with each relation value found in the input frame. It was tested extensively using print statements. After the resolve relations routine was verified, the determine units routine was written.

The determine units routine is a recursive routine, which locates the units for a given relation value. After it was completed, it was tested using print statements to monitor the replacement process. Once the process was verified, the and units routine was created to complete the replacement process. This routine was also tested using print statements.

The compare units routine was written after the replacement process had been verified. This routine verifies each output and-set. It was first written using only the input and-sets. After these sets were being compared properly, the uses and-sets were added.

Next, the synonym replacement code was added, and finally the conversion code was added. Each step was verified by using print statements. Once one step was completely tested, the next step was begun.

The last routine to be developed was the print errors routine. This routine is very simple. It is responsible for printing either the short or the long report form. It was verified by requesting the long form and then the short form. The output printed for each report type was verified.

In general, each routine was tested by printing intermediate data. Each data structure was printed as it was being populated and finally, each data structure was printed after it was complete. In addition, each data structure was verified for array limits, since C does not check array bounds. Very little error checking was performed on the input specification itself, since it is assumed to have been machine generated and to have gone through various completeness checks before it is passed to the units of measure consistency checker. Extensive error checking was done on the consistency checking of the code. Sample specifications, in which units were missing, unresolved, required synonyms or required conversions were used during testing. The program was able to locate and identify consistency errors (See Appendix E for some sample output).

Chapter Five

Conclusions and Extensions

The units of measure consistency checker is not very valuable alone. It was created to be used in conjunction with the software development environment currently being prototyped at Kansas State University. The units of measure consistency checker described here, along with several other types of consistency and correctness verification systems, will be extremely valuable in locating errors within the requirements specification documents created within this new environment. By locating these errors in the requirements stage of the development cycle, more costly errors can be avoided further along in the development cycle.

The design of the units of measure consistency checker is extremely simple. It is based on the assumption that the checker would be most valuable in analyzing extremely large requirements specification documents, and that it is part of a more complex system. Therefore, it was designed to use a minimum of resources. The checker uses a file structure to store the parsed specification. If the use of resources is of less importance, several changes could be made to the units of measure consistency checker to increase its speed and make it more dynamic.

In order to improve the checker's ability to handle large complex requirements specifications, the array structures used in the checker could be changed to linked lists. This change would eliminate the possibility of running out of room in the array structures. The linked lists would also save space when complex relations were not being analyzed. If this change were incorporated, the complexity of storing the parsed frames in the file structures, would increase greatly.

In order to improve the speed of the units of measure consistency checker, the parser and the checker could be rewritten as one program, keeping the parsed specification in memory. The i/o data structures could be stored in four sorted trees. The determine un-

its routine would then use these structures to locate the appropriate i/o data frames to resolve the relations in the function frames. The rest of the program would remain the same.

In addition to the design changes outlined in the paragraphs above, several extensions could be made to the current capabilities of the units of measure consistency checker. The checker processes "and" and "or" conditions found within the text of the relations values. The parsing of the relations values could be expanded to recognize combinations of "and" and "or" conditions using parenthesis. The compare units routine could also be expanded to recognize the "and" and "or" conditions within the uses relations. Finally, the units of measure consistency checker could also be expanded to provide an exact cancellation of units.

In general, the area of mechanized requirements specifications is still very new. The units of measure consistency checker, together with other forms of consistency checking, could provide great cost savings in the software development life cycle.

Bibliography

[1] Abbot, Russell J. "Program Design by Informal English Descriptions." Communications of the ACM, Vol. 26, No. 11, November 1983, pp. 883-894.

[2] Agusa, Kiyoshi, Atsushi Ohnishi, and Yutaka Ohno. "Verification System for Formal Requirements Description." IEEE, 1982, pp. 120-126.

[3] Balzer, Robert and Neil Goldman. "Principles of Good Software Specifications and Their Implications for Specification Languages." National Computer Comference, 1981, pp. 393-400.

[4] Buaer, Friedrich L. "From Specifications to Machine Code: Program Construction through Formal Reasoning."IEEE, 1982, pp. 84-91.

[5] Belford, P. C. and D. S. Taylor. "Specification Verification - A Key to Improving Software Reliability." Computer Software Engineering, April 1976, pp. 83-96.

[6] Boehm, Barry W. "Verifying and Validating Software Requirements ans Design Specifications." IEEE Software, Vol. 1, No. 1, January 1984, pp. 75-88.

[7] Boehm, B. W., R. K. McClean and D. B. Urfrig. "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software." Proceedings, International Conference on Reliable Software, April 1975, pp. 105-113.

[8] Campbell, R. H. and P. G. Richards. "SAGA: A System to Automate the Management of Software Production." National Computer Conference, 1981, pp. 231-234.

[9] Chen, Peter Pin-Shan, "The Entity-Relationship Model- Toward a Unified View of Data." ACM Transactions on Data Base Systems, Vol. 1, No. 1, March 1976, pp. 9-36.

[10] Davis, Alan M. and Tomlinson G. Rauscher. "Formal Techiniques and Automatic Processing to Ensure Correctness in Requirements Specifications." Proceedings, Specifications of Reliable Software, 1979, pp. 15-35.

[11]Greenspan, Sol J., John Mylopoulos and Alex Borgida. "Capturing More World Knowledge in the Requirements Specification." 6th International Conference on Software Engineering, September 1982, pp. 225-234.

[12] Heitmeyer, Constance 1. and John D. McLean. "Abstract Requirements Specifications: A New Approach and its Application." IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, September 1983, pp. 580-589.

[13] Matsumoto, Yoshihro and Kazuo Matsumuro. "A Specification Analysis and Documentation System for Process Control Software." The IEEE Computer Society's Fifth International Computer Software and Application Conference, 1981, pp. 411-417.

[14] McCorduck, Pamela. "Introduction to the Fifth Generation." Communications of the ACM, Vol. 26, No. 9, September 1983, pp. 629-630.

[15] Nyari, Erika and Harry Sneed. "Sofspec: A Pragmatic Approach to Automated Specification Verification." The Journal of Systems and Software, 3, 1983, pp.

193-200.

[16] Reifer, Donald J. and Stephen Trattner. "A Glossary of Software Tools and Techniques." Computer, July 1977, pp. 6-14.

[17] Riddle, William E. "An Assessment of Dream." Tutorial on Software Systems, 1981, pp. 231-234.

[18] Shapiro, Ehud Y. "The Fifth Generation Project - A Trip Report." Communications of the ACM, Vol. 26, No. 9, September 1983, pp. 637-641.

[19] Stephens, Sharon A. and Leonard L. Tripp. "Requirements Expression and Verification Aid." Proceedings, 3rd International Conference on Software Engineering, May 1978, pp. 60-67.

[20] Teichroew, Daniel and Ernest a. Hershey,III. "PSL/ PSA: A Computer - Aided Technique for Structured Documentation and Analysis of Information Processing Systems." IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 41-48.

[21] Tichy, Walter F. "Software Development Control Based on Module Interconnection." Proceedings, 4th International Conference on Software Engineering, 1979, pp. 29-41.

[22] Treleaven, P. C. and I. Gouveia Lima. "Japan's Fifth Generation Computer System." Technical Report Series, University of Newcastle upon Tyne, May 1982, No. 176.

[23] Wasserman, Anthony I. "Toward Integrated Software Development Environments." IEEE, 1981, pp. 15-35.

[24] Zave, Pamela. "The Operational Versus the Conventional Approach to Software Development." Communications of the ACM, Vol. 27, No. 2, February 1984, pp. 104-118.

[25] Bell, Thomas E., David C. Bixler, and Margret E. Dyer. "An Extendable Approach to Computer - Aided Software Requirements Engineering." IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 49-60.

[26] Ross, Douglas T. and Kenneth E. Schoman, Jr. "Structured Analysis for Requirements Definition." IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 6-15.

[27] Ross, Douglas T. "Structured Analysis (SA): A Language for Communicating Ideas." IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 16-34.

Appendix A

BNF Syntax Description

```
<era_spec> ::=
  <era_title> <era_body> <mode_table>

<era_title> ::=
  PROCESS : <text>

<era_body> ::=
  <frame> | <frame> <era_body>

<frame> ::=
  <NL> <NL> <frame_header> <frame_body>
  | <NL> <NL> Comment : <text_lines>

<frame_header> ::=
  <i_o_data_header> : <i_o_data_name>
  | <function_header> : <CAPITAL_WORD>

<i_o_data_header> ::=
  Type | Input | Output | Input_output | Data
  | Constant | <CAPITAL_WORD>

<function_header> ::=
  Activity | Periodic_function | <CAPITAL_WORD>

<frame_body> ::=
  <relation> | <relation> <frame_body>

<relation> ::=
  <NL_B> <relation_type> : <relation_value>

<relation_type> ::=
  keywords | input | output | required_mode
  | necessary_condition | occurence | assertion
  | action | comment | media | structure | type
  | units | subpart_is | subpart_of | uses
  | <WORD>

<relation_value> ::=
```

```
  <text_lines> | <structure>

<structure> ::=
  <struct> | <struct> <NL_B> : <structure>

<struct> ::=
  <name> | <text> | <name> <structure>
  | <text> <structure>

<name> ::=
  <mode_name> | <i_o_data_name>

<i_o_data_name> ::=
  $ <WORD> $

<mode_name> ::=
  * <WORD> *

<mode_table> ::=
  <NL> <NL> MODE_TABLE <mode_list> <inital_mode>
  <transition_body>

<mode_list> ::=
  <mode> | <mode> <mode_list>

<mode> ::=
  <NL_B> Mode : <mode_name>

<initial_mode> ::=
  <NL> <NL_B> Initial_Mode : <mode_name>

<transition_body> ::=
  <NL> <NL_B> Allowed_Mode_Transitions :
  <transition_list>

<transition_list> ::=
  <transition> | <transition> <transition_list>

<transition> ::=
  <NL_B> <event> : <mode_name> -> <mode_name>
```

<event> ::=
  <i_o_data_name> | <i_o_data_name> = ' <text> '
  | <function_header>

<text_lines> ::=
  <text> | <text> <text_cont>

<text> ::=
  <WORD> | <WORD> <text>

<text_cont> ::=
  <NL_B> : <text> | <NL_B> : <text> <text_cont>

<NL> ::=
  '0 | '0 <NL>

<NL_B> ::=
  <NL> ' '

LEXICAL SCANNER INFORMATION

Tokens used in the productions above begin with
one of the following:
*,$, .',:,-,=,<char>,{,} (excluding the
commas)

The following tokens are important above:
<WORD> ::= <char> | <char> <WORD>
<CAPITAL_WORD> ::= <capital_letter> <WORD>

<char> ::=
  <lower_case_char> | <symbol>

<lower_case_char> ::=
  a | b | ... | z | 0 | 1 | ... | 9

<symbol> ::=
  # | % | & | (|) | ? | _

<capital_letter> ::=

A | B | ... | Z

There exists a set of "reserved word" tokens which includes:

{keyboard,crt,internal,secondary_storage,
NONE,every,mode}

Appendix B

Sample E-R-A Requirements Specification

PROCESS $weather_calc$

Activity : $predict_arrival$
  input : $current_position$
  input : $500mb_winds$
  output : $arrival_time$

Activity : $predict_wind_velocity$
  input : $surface_pressure_gradient$
  uses   : $hydrostatic_parameter$
  output : $surface_wind$

Activity : $predict_low_temp$
  input : $skewT_temp$
  output : $low_prediction$

Activity : $predict_precip$
  input : $upper_air_moisture$
  uses   : $water_density$
  output : $moisture_total$

Input   : $current_position$
  type : integer
  units : miles

Input : $surface_pressure_gradient$

type : real

units : millibars / kilometer

Input : $skewT_data$

type : array of $temp$

size : 1..100

Input : $upper_air_moisture$

type : array of $specific_humidity$

size : 1..100

Input : $500mb_winds$

type : $velocity$

Output : $moisture_totals$

type : real

units : centimeters

Type : $velocity$

structure : $speed$, $direction$

Type : $speed$

type : real

units : meters/second

Type : $direction$

type : integer

units : degrees

Output : $arrival_time$

type : char

units : hours and minutes

Output : $surface_wind$

type : real

units : meters/second

Output : $low_prediction$

type : $temp$

Type : $temp$

type : real

units : degrees Celsius

Type : $specifc_humidity$

type : real

units : grams/centimeter**3

Constant : $hydrostatic_parameter$

type : real

units : meters**2/seconds

Constant : $water_density$

type : real

value : 1

units : grams/centimeter**3

Appendix C.1

Function Frame Data Structure

```
#define WORD_SIZE 50

#define FRAME_OR_MAX 10
#define FRAME_AND_MAX 10

struct rel_val
    {
    char term[WORD_SIZE+1];
    };

struct fstart
    {
    char ftype[WORD_SIZE+1];
    char name[WORD_SIZE+1];
    };

struct fframe
    {
    struct fstart header;
    struct rel_val
        input[FRAME_OR_MAX+1][FRAME_AND_MAX+1];
    struct rel_val
        output[FRAME_OR_MAX+1][FRAME_AND_MAX+1];
```

```
struct rel_val
    uses[FRAME_OR_MAX+1][FRAME_AND_MAX+1];
};
```

Appendix C.2

I/O Data Frame Data Structure

```
#define WORD_SIZE 50

#define FRAME_OR_MAX 10
#define FRAME_AND_MAX 10

struct rel_val
    {
    char term[WORD_SIZE+1];
    };

struct fstart
    {
    char ftype[WORD_SIZE+1];
    char name[WORD_SIZE+1];
    };

struct iframe
    {
    struct fstart header;
    struct rel_val
        structure[FRAME_OR_MAX+1][FRAME_AND_MAX+1];
    struct rel_val
        type[FRAME_OR_MAX+1][FRAME_AND_MAX+1];
```

```
struct rel_val
    units[FRAME_OR_MAX+1][FRAME_AND_MAX+1];
};
```

```
#define WORD_SIZE 50

#define FRAME_OR_MAX 10

#define FRAME_AND_MAX 10

#define AND_SET_MAX 30

struct rel_val
    char term[WORD_SIZE+1];
    };

struct and_set
    struct rel_val units[AND_SET_MAX+1];
    struct and_set *or_set;
    };

struct hframe
    struct and_set *input[FRAME_OR_MAX+1];
    struct and_set *output[FRAME_OR_MAX+1];
    struct and_set *uses[FRAME_OR_MAX+1];
    };
```

Resolved Relations Frame Data Structure

Appendix D.1

Sample FUNCT Data File

```
Activity : $predict_arrival$
  input  : $current_position$
  input  : $500mb_winds$
  output : $arrival_time$

Activity : $predict_wind_velocity$
  input  : $surface_pressure_gradient$
  output : $surface_wind$
  uses   : $hydrostatic_parameter$

Activity : $predict_low_temp$
  input  : $skewT_temp$
  output : $low_prediction$

Activity : $predict_precip$
  input  : $upper_air_moisture$
  output : $moisture_total$
  uses   : $water_density$
```

Appendix D.2

Sample INDATA Data File

```
Input   : $current_position$
 units  : miles

Input   : $surface_pressure_gradient$
 units  : millibars kilometer

Input   : $skewT_data$
 type   : $temp$

Input   : $upper_air_moisture$
 type   : $specific_humidity$

Input   : $500mb_winds$
 type   : $velocity$
```

Appendix D.3

Sample OTDATA Data File

```
Output : $moisture_totals$
  units  : centimeters

Output : $arrival_time$
  units  : hours minutes

Output : $surface_wind$
  units  : meters second

Output : $low_prediction$
  type   : $temp$
```

Appendix D.4

Sample TDATA Data File

```
Type      : $velocity$
 structure : $speed$  $direction$

Type      : $speed$
 units    : meters second

Type      : $direction$
 units    : degrees

Type      : $temp$
 units    : degrees Celsius

Type      : $specifc_humidity$
 units    : grams centimeter
```

Appendix D.5

Sample UDATA Data File

```
Constant : $hydrostatic_parameter$
  units  : meters seconds

Constant : $water_density$
  units  : grams centimeter
```

Appendix E.1

Long Output

```
Processing file spec.uts
report type = long

Activity : $predict_arrival$
Processing output set -
    $arrival_time$
    hours minutes
Processing input set -
    $current_position$
    miles
  Warning - unresolved units - hours minutes
Processing input set -
    $500mb_winds$
    meters second degrees

Activity : $predict_wind_velocity$
Processing output set -
    $surface_wind$
    meters second
Processing input set -
    $surface_pressure_gradient$
    millibars kilometer
  Warning - conversion required second
```

Activity : $predict_low_temp$

Processing output set -

$low_prediction$

degrees celsius

Processing input set -

$skewT_temp$

no match

Warning - unresolved units - degrees celsius

Error - output units set was never produced

Activity : $predict_precip$

Processing output set -

$moisture_total$

no match

Processing input set -

$upper_air_moisture$

no match

Appendix E.2

Short Output

```
Processing file spec.uts
report type = short

Activity : $predict_arrival$
Processing output set -
    $arrival_time$
    hours minutes
Processing input set -
    $current_position$
    miles
  Warning - unresolved units - hours minutes
Processing input set -
    $500mb_winds$
    meters second degrees

Activity : $predict_wind_velocity$
Processing output set -
    $surface_wind$
    meters second
Processing input set -
    $surface_pressure_gradient$
    millibars kilometer
  Warning - conversion required second
```

Activity : $predict_low_temp$

Processing output set -

$low_prediction$

degrees celsius

Processing input set -

$skewT_temp$

no match

Warning - unresolved units - degrees celsius

Error - output units set was never produced

Appendix F

```
: 'This shell procedure preprocesses the'
: 'input files and calls the units checker'

: 'Process options'

report=short
while test TRUE
do
        case ${1} in
        -l*)
                : 'Long report form requested'
                report=long
                ;;
                : 'Process illegal option'
                echo illegal option ${1}
                ;;
                : 'No more options, begin'
                : 'main processing'
                break
                ;;
        esac
        shift
done

: 'Start main processing'
```

```
if test $# -eq 0
then
  echo "Input file needed"
else
  while test $# -ne 0
  do
    if test -r "$1"
    then
        echo "Processing file $1"
        awk -f pre.cont $1 |
        awk -f pre.strip > $1.temp
        units.parser $1.temp
        rm $1.temp
        units.checker $report
    else
        echo "Can't open $1"
    fi
    shift
  done
fi
```

Scanner

```
Process continue statements

$1 !~ /:/ {print;prev=$1}

$1 ~ /:/ {$1 = prev" + ";print}

Strip out unnecessary entities and

relations/attributes

$1 ~ /^Activity$/ {print"";print}

$1 ~ /^Periodic_function$/ {print"";print}

$1 ~ /^Type$/ {print"";print}

$1 ~ /^Input$/ {print"";print}

$1 ~ /^Output$/ {print"";print}

$1 ~ /^Input_output$/ {print"";print}

$1 ~ /^Data$/ {print"";print}

$1 ~ /^Constant$/ {print"";print}

$1 ~ /^input$/ {print}

$1 ~ /^output$/ {print}

$1 ~ /^type$/ {print}

$1 ~ /^units$/ {print}

$1 ~ /^uses$/ {print}

$1 ~ /^structure$/ {print}
```

Word Data Files

/* Double Words */

degrees celsius

degrees farenheit

/* Synonyms */

second : seconds

second : sec

seconds : second

seconds : sec

sec : second

sec : seconds

/* Conversions */

second : minute

second : minutes

second : min

second : hour

second : hours

seconds : minute

seconds : minutes

seconds : min

seconds : hour

seconds : hours

sec : minute

sec : minutes

sec : min

sec : hour

sec : hours

minute : second

minute : seconds

minute : sec

minutes : second

minutes : seconds

minutes : sec

min : second

min : seconds

min : sec

hour : second

hour : seconds

hour : sec

hours : second

hours : seconds

hours : sec

Header File For C - Code

```
#define FRAME_SEPERATOR '0

#define OR ':'
#define AND '+'
#define DELM '$'

#define WORD_SIZE 50
#define LINE_SIZE 80

#define FRAME_OR_MAX 10
#define FRAME_AND_MAX 10

#define AND_SET_MAX 30

#define PMODE 0600

struct rel_val
    {
    char term[WORD_SIZE+1];
    };

struct fstart
    {
    char ftype[WORD_SIZE+1];
    char name[WORD_SIZE+1];
```

```
};

struct fframe
    {
    struct fstart header;
    struct rel_val input
        [FRAME_OR_MAX+1][FRAME_AND_MAX+1];
    struct rel_val output
        [FRAME_OR_MAX+1][FRAME_AND_MAX+1];
    struct rel_val uses
        [FRAME_OR_MAX+1][FRAME_AND_MAX+1];
    };

struct iframe
    {
    struct fstart header;
    struct rel_val structure
        [FRAME_OR_MAX+1][FRAME_AND_MAX+1];
    struct rel_val type
        [FRAME_OR_MAX+1][FRAME_AND_MAX+1];
    struct rel_val units
        [FRAME_OR_MAX+1][FRAME_AND_MAX+1];
    };

struct and_set
    {
    struct rel_val units[AND_SET_MAX+1];
```

```
    struct and_set *or_set;
    };

struct hframe
    {
    struct and_set *input[FRAME_OR_MAX+1];
    struct and_set *output[FRAME_OR_MAX+1];
    struct and_set *uses[FRAME_OR_MAX+1];
    };
```

Parser

```
#include <stdio.h>
#include <ctype.h>
#include "era.parser.c"

main(argc,argv)
int argc;
char *argv[];

{
FILE *ifp,      /* input file pointer      */
    *fopen();

if ((ifp = fopen(*++argv,"r")) == NULL)
  {
  printf("Can't open %s0, *argv);
  }
else
  {
  det_ftype(ifp);
  fclose(ifp);
  }
}
```

Determine Frame Type

```
#include <stdio.h>
#include <ctype.h>
#include "era.def"
#include "era.struct"
#include "era.funct.c"
#include "era.iod.c"

det_ftype(ifp)
FILE *ifp;          /* Input file pointer */

{
int ffd,  /* Function Data file descriptor */
   infd,          /* Input Data file descriptor    */
   otfd,          /* Output Data file descriptor   */
   tfd,  /* Type Data file descriptor     */
   ufd;  /* Uses Data file descriptor     */

struct fframe funct;
struct iframe iod;

int i,             /* String index */
   j;              /* String index */

char line[LINE_SIZE],
    token[WORD_SIZE];
```

```
infd = creat("INDATA", PMODE);
otfd = creat("OTDATA", PMODE);
tfd = creat("TDATA", PMODE);
ufd = creat("UDATA", PMODE);
ffd = creat("FUNCT", PMODE);

if (infd == -1 || otfd == -1 || tfd == -1
                || ufd == -1 || ffd == -1)
  prinf("Error *** unable to create DATA files0);
else
  {
  /* parse era spec */
  while (fgets(line,LINE_SIZE,ifp) != NULL)
   {
   if (line[0] != FRAME_SEPERATOR)
       {
       /* parse era frame */
       i = -1;
       while (isspace(line[++i]) != 0
                   && line[i] != ' ');
       if (isupper(line[i]) != 0)
         {
         /* first character of frame */
         /* type is capital          */

         j = 0;
         token[j] = line[i];
```

```
while ((islower(line[++i]) != 0
                || line[i] == '_' )
                && line[i] != ' ')
  token[++j] = line[i];
token[++j] = ' ';

/* determine entity type */

if (strcmp(token,"Activity") == 0)
  {
  /* function entity */
  parse_funct(line,ifp,&funct);
  write(ffd,&funct,sizeof(funct));
  }
else

if (strcmp(token,"Periodic_function") == 0)
  {
  /* function entity */
  parse_funct(line,ifp,&funct);
  write(ffd,&funct,sizeof(funct));
  }
else

if (strcmp(token,"Input") == 0)
  {
  /* I_O_Data entity */
```

```
  parse_iod(line,ifp,&iod);
  write(infd,&iod,sizeof(iod));
  }
else

if (strcmp(token,"Output") == 0)
  {
  /* I_O_Data entity */
  parse_iod(line,ifp,&iod);
  write(otfd,&iod,sizeof(iod));
  }
else

if (strcmp(token,"Input_output") == 0)
  {
  /* I_O_Data entity */
  parse_iod(line,ifp,&iod);
  write(infd,&iod,sizeof(iod));
  write(otfd,&iod,sizeof(iod));
  }
else

if (strcmp(token,"Type") == 0)
  {
  /* I_O_Data entity */
  parse_iod(line,ifp,&iod);
  write(tfd,&iod,sizeof(iod));
```

```
    }
  else

  if (strcmp(token,"Constant") == 0)
    {
    /* I_O_Data entity */
    parse_iod(line,ifp,&iod);
    write(ufd,&iod,sizeof(iod));
    }
  else

  if (strcmp(token,"Data") == 0)
    {
    /* I_O_Data entity */
    parse_iod(line,ifp,&iod);
    write(ufd,&iod,sizeof(iod));
    }
  else
    {
    /* entity type was added to preprocessor */
    /* but not to parser                     */
    printf("*** Warning *** unknown entity");
    printf("*** line flushed0);
    printf("%s",line);
    }
  }
else
```

```
            {
            /* first character is not upper case  */
            /* flush rest of unknown entity frame */
            printf("*** Warning *** line flushed0);
            printf("%s",line);
            }
          }
    }
  close(infd);
  close(otfd);
  close(tfd);
  close(ufd);
  close(ffd);
  }
}
```

Parse Function Frame

```
#include <stdio.h>
#include <ctype.h>
#include "era.def"

parse_funct(line,ifp,funct)
char line[];
FILE *ifp;    /* Input file pointer        */
struct fframe *funct;

{
int i,
    j;

int input_or_index,
    input_and_index,
    output_or_index,
    output_and_index,
    uses_or_index,
    uses_and_index;

char token[WORD_SIZE];

/* process function frame header */
i = -1;
while (isspace(line[++i]) != 0
```

```
            && line[i] != ' ');

j = 0;
token[j] = line[i];
while ((islower(line[++i]) != 0
        || line[i] == '_') && line[i] != ' ')
  token[++j] = line[i];
token[++j] = ' ';

strcpy(funct->header.ftype,token);

while ((isspace(line[++i]) != 0
        || line[i] == ':') && line[i] != ' ');

j = 0;
token[j] = line[i];
while (isspace(line[++i]) == 0 && line[i] != ' ')
  token[++j] = line[i];
token[++j] = ' ';

strcpy(funct->header.name,token);

/* process function frame body */
strcpy(funct->input[0][0].term,"");
strcpy(funct->output[0][0].term,"");
strcpy(funct->uses[0][0].term,"");
```

```
input_or_index = -1;
output_or_index = -1;
uses_or_index = -1;

while ((fgets(line,LINE_SIZE,ifp) != NULL)
        && (line[0] != FRAME_SEPERATOR))
 {
 /* skip beginning white space */
 i = -1;
 while (isspace(line[++i]) != 0
          && line[i] != ' ');

 /* process text */
 if (islower(line[i]) != 0)
   {
   /* first character of relation/attribute */
   /* is lower case                        */

   j = 0;
   token[j] = line[i];
   while ((islower(line[++i]) != 0
         || line[i] == '_') && line[i] != ' ')
    token[++j] = line[i];
   token[++j] = ' ';

   /* determine relation/attribute type */
```

```
if (strcmp(token,"input") == 0)
    {
    /* parse input relation */

    /* skip white space */
    while (isspace(line[++i]) != 0
                && line[i] != ' ');

    while (line[i] != ' ')
     {
     if (line[i] == OR)
       {
       /* set up or index */
       input_and_index = 0;
       if (input_or_index == -1)
            input_or_index++;
       if (strcmp(funct->input[input_or_index]
              [input_and_index].term, "") != 0)
            input_or_index++;
       if (input_or_index < FRAME_OR_MAX)
            strcpy(funct->input[input_or_index+1]
                     [input_and_index].term, "");
       }

     /* parse relation/attribute value */
     if (input_or_index < FRAME_OR_MAX)
       {
```

```
/* find start of nonterminal */
while (line[++i] != DELM
        && line[i] != ' ');

if (line[i] == DELM)
    {
    j = 0;
    token[j] = line[i];
    while (line[++i] != DELM
          && line[i] != ' ')
      token[++j] = line[i];
    if (line[i] == DELM)
      {
      token[++j] = line[i];
      token[++j] = ' ';

      /* store nonterminal */
      if (input_and_index < FRAME_AND_MAX)
        {
        strcpy(funct-> input[input_or_index]
                [input_and_index].term.token);
        input_and_index++;
        strcpy(funct-> input[input_or_index]
                [input_and_index].term, "");
        }
      else
        {
```

```
            printf("*** Error *** AND limit reached");
            printf(" *** function parser0);
            printf("%s : %s0, funct-> header.ftype,
                    funct-> header.name);
            printf("%s", line);
            }
          }
        else
          {
          printf("*** Error *** invalid relation ");
          printf("%s0, token);
          }
        }
    else
        if ((line[i] == 'o' || line[i] == 'O')
           && (line[i+1] == 'r' || line[i+1] == 'R'))
          {
          line[i] = OR;
          line[i+1] = ' ';
          }
    }
  }
else
  {
  printf("*** Error *** OR limit reached ");
  printf("*** function parser0);
  printf("%s : %s0, funct-> header.ftype,
```

```
                funct->header.name);
        }
    }
else

if (strcmp(token,"output") == 0)
    {
    /* parse output relation */

    /* skip white space */
    while (isspace(line[++i]) != 0
                && line[i] != ' ');

    while (line[i] != ' )
     {
     if (line[i] == OR)
       {
       /* set up or index */
       output_and_index = 0;
       if (output_or_index == -1)
            output_or_index++;
       if (strcmp(funct->output[output_or_index]
                [output_and_index].term, "") != 0)
            output_or_index++;
       if (output_or_index < FRAME_OR_MAX)
            strcpy(funct->output[output_or_index+1]
                        [output_and_index].term, "");
```

```
  }

/* parse relation/attribute value */
if (output_or_index < FRAME_OR_MAX)
  {
  /* find start of nonterminal */
  while (line[++i] != DELM
         && line[i] != ' ');

  if (line[i] == DELM)
      {
      j = 0;
      token[j] = line[i];
      while (line[++i] != DELM
           && line[i] != ' ')
        token[++j] = line[i];
      if (line[i] == DELM)
        {
        token[++j] = line[i];
        token[++j] = ' ';

        /* store nonterminal */
        if (output_and_index < FRAME_AND_MAX)
          {
          strcpy(funct->output[output_or_index]
                 [output_and_index].term,token);
          output_and_index++;
```

```
            strcpy(funct->output[output_or_index]
                        [output_and_index].term, "");
            }
          else
            {
            printf("*** Error *** AND limit reached");
            printf(" *** function parser0);
            printf("%s : %s0, funct->header.ftype,
                    funct->header.name);
            printf("%s", line);
            }
          }
        else
          {
          printf("*** Error *** invalid relation ");
          printf("%s0, token);
          }
        }
      else
        if ((line[i] == 'o' || line[i] == 'O')
            && (line[i+1] == 'r' || line[i+1] == 'R'))
          {
          line[i] = OR;
          line[i+1] = ' ';
          }
    }
}
```

```
    else
      {
      printf("*** Error *** OR limit reached ");
      printf("*** function parser0);
      printf("%s : %s0, funct-> header.ftype,
              funct-> header.name);
      }
    }
else

if (strcmp(token,"uses") == 0)
    {
    /* parse uses relation */
    while (isspace(line[++i]) != 0
              && line[i] != ' ');

    while (line[i] != ' ')
     {
     if (line[i] == OR)
       {
       /* set up or index */
       uses_and_index = 0;
       if (uses_or_index == -1)
           uses_or_index++;
       if (strcmp(funct-> uses[uses_or_index]
              [uses_and_index].term, "") != 0)
           uses_or_index++;
```

```
  if (uses_or_index < FRAME_OR_MAX)
      strcpy(funct->uses[uses_or_index+1]
            [uses_and_index].term, "");
  }

/* parse relation/attribute value */
if (uses_or_index < FRAME_OR_MAX)
  {
  /* skip white space */
  while (line[++i] != DELM
        && line[i] != ' ');

  /* build nonterminal */
  if (line[i] == DELM)
      {
      j = 0;
      token[j] = line[i];
      while (line[++i] != DELM
            && line[i] != ' ')
        token[++j] = line[i];
      if (line[i] == DELM)
        {
        token[++j] = line[i];
        token[++j] = ' ';

        /* store nonterminal */
        if (uses_and_index < FRAME_AND_MAX)
```

```
            {
            strcpy(funct-> uses[uses_or_index]
                    [uses_and_index].term, token);
            uses_and_index++;
            strcpy(funct-> uses[uses_or_index]
                    [uses_and_index].term, "");
            }
          else
            {
            printf("*** Error *** AND limit reached");
            printf(" *** function parser0);
            printf("%s : %s0, funct-> header.ftype,
                    funct-> header.name);
            printf("%s", line);
            }
          }
        else
          {
          printf("*** Error *** invalid relation ");
          printf("%s0, token);
          }
        }
      else
        if ((line[i] == 'o' || line[i] == 'O')
            && (line[i+1] == 'r' || line[i+1] == 'R'))
          {
          line[i] = OR;
```

```
                line[i+1] = ' ';
                }
            }
          }
        else
          {
          printf("*** Error *** OR limit reached ");
          printf("*** function parser0);
          printf("%s : %s0, funct-> header.ftype,
                funct-> header.name);
          }
        }
    else
        {
        /* preprocessor will also allow structure, type */
        /* and uses relations/attributes, however they        */
        /* are parsed by the iod frame parser.          */
        printf("*** Warning *** unknown function");
        printf(" relation/attribute0);
        printf("*** line flushed *** function parser0);
        printf("%s",line);
        }
    }
  }
}
```

Parse I/O Data Frame

```
#include <stdio.h>
#include <ctype.h>
#include "era.def"

parse_iod(line,ifp,iod)
char line[];
FILE *ifp;        /* Input file pointer        */
struct iframe *iod;

{
int i,
    j;

int type_or_index,
    type_and_index,
    str_or_index,
    str_and_index,
    uts_or_index,
    uts_and_index;

int match;

char word1[WORD_SIZE],
     word2[WORD_SIZE],
     double_word[LINE_SIZE];
```

```
char token[WORD_SIZE];

FILE *fopen(),
    *dwfp;

/* process iod frame header */
i = -1;
while (isspace(line[++i]) != 0
            && line[i] != ' ');

j = 0;
token[j] = line[i];
while ((islower(line[++i]) != 0
        || line[i] == '_') && line[i] != ' ')
  token[++j] = line[i];
token[++j] = ' ';

strcpy(iod-> header.ftype.token);

while (line[++i] != DELM
      && line[i] != ' ');

if (line[i] == DELM)
  {
  j = 0;
  token[j] = line[i];
  while (line[++i] != DELM
```

```
        && line[i] != ' ')
    token[++j] = line[i];
  if (line[i] == DELM)
    {
    token[++j] = line[i];
    token[++j] = ' ';
    strcpy(iod-> header.name,token);
    }
  else
    printf("%s invalid nonterminal0,token);
  }
else
  printf("frame name not found0);

/* process iod frame body */

strcpy(iod-> structure[0][0].term,"");
strcpy(iod-> type[0][0].term,"");
strcpy(iod-> units[0][0].term,"");

type_or_index = -1;
str_or_index = -1;
uts_or_index = -1;

while(fgets(line,LINE_SIZE,ifp) != NULL
        && line[0] != FRAME_SEPERATOR)
 {
```

```
/* skip white space */
i = -1;
while (isspace(line[++i]) != 0
                && line[i] != '0');
/* process text */
if (islower(line[i]) != 0)
  {
  /* first character of relation/attribute */
  /* is lower case                        */

  j = 0;
  token[j] = line[i];
  while ((islower(line[++i]) != 0
          || line[i] == '_') && line[i] != ' ')
   token[++j] = line[i];
  token[++j] = ' ';

  /* determine relation/attribute type */

  if (strcmp(token,"structure") == 0)
      {
      /* parse structure relation */

      /* skip white space */
      while (isspace(line[++i]) != 0
                  && line[i] != ' ');
```

```
while (line[i] != ' ')
 {
 if (line[i] == OR)
   {
   /* set up or index */
   str_and_index = 0;
   if (str_or_index == -1)
       str_or_index++;
   if (strcmp(iod->structure[str_or_index]
         [str_and_index].term, "") != 0)
       str_or_index++;
   if (str_or_index < FRAME_OR_MAX)
       strcpy(iod->structure[str_or_index+1]
             [str_and_index].term, "");
   }

 /* parse relation/attribute value */
 if (str_or_index < FRAME_OR_MAX)
   {
   /* find start of nonterminal */
   while (line[++i] != DELM
         && line[i] != ' ');

   if (line[i] == DELM)
       {
       j = 0;
       token[j] = line[i];
```

```
while (line[++i] != DELM
      && line[i] != ' ')
  token[++j] = line[i];
if (line[i] == DELM)
  {
  token[++j] = line[i];
  token[++j] = ' ';

  /* store nonterminal */
  if (str_and_index < FRAME_AND_MAX)
    {
    strcpy(iod->structure[str_or_index]
            [str_and_index].term, token);
    str_and_index++;
    strcpy(iod->structure[str_or_index]
            [str_and_index].term, "");
    }
  else
    {
    printf("*** Error *** AND limit reached");
    printf(" *** iod parser0);
    printf("%s : %s0, iod->header.ftype,
            iod->header.name);
    printf("%s", line);
    }
  }
else
```

```
                {
                printf("*** Error *** invalid relation ");
                printf("%s0, token);
                }
            }
        else
            if ((line[i] == 'o' || line[i] == 'O')
                && (line[i+1] == 'r' || line[i+1] == 'R'))
                {
                line[i] = OR;
                line[i+1] = ' ';
                }
        }
      }
    else
      {
      printf("*** Error *** OR limit reached ");
      printf("*** iod parser0);
      printf("%s : %s0, iod-> header.ftype,
              iod-> header.name);
      }
    }
else

if (strcmp(token,"type") == 0)
    {
    /* parse type relation */
```

```
/* skip white space */
while (isspace(line[++i]) != 0
        && line[i] != ' ');

while (line[i] != ' ')
 {
 if (line[i] == OR)
   {
   /* set up or index */
   type_and_index = 0;
   if (type_or_index == -1)
       type_or_index++;
   if (strcmp(iod-> type[type_or_index]
           [type_and_index].term, "") != 0)
       type_or_index++;
   if (type_or_index < FRAME_OR_MAX)
       strcpy(iod-> type[type_or_index+1]
               [type_and_index].term, "");
   }

 /* parse relation/attribute value */
 if (type_or_index < FRAME_OR_MAX)
   {
   /* find start of nonterminal */
   while (line[++i] != DELM
           && line[i] != ' ');
```

```
if (line[i] == DELM)
    {
    j = 0;
    token[j] = line[i];
    while (line[++i] != DELM
        && line[i] != ' ')
      token[++j] = line[i];
    if (line[i] == DELM)
      {
      token[++j] = line[i];
      token[++j] = ' ';

      if (type_and_index < FRAME_AND_MAX)
        {
        strcpy(iod-> type[type_or_index]
                [type_and_index].term, token);
        type_and_index++;
        strcpy(iod-> type[type_or_index]
                    [type_and_index].term, "");
        }
      else
        {
        printf("*** Error *** AND limit reached");
        printf(" *** iod parser0);
        printf("%s : %s0, iod-> header.ftype,
                iod-> header.name);
        printf("%s", line);
```

```
                }
              }
            else
              {
              printf("*** Error *** invalid relation ");
              printf("%s0, token);
              }
            }
          else
            if ((line[i] == 'o' || line[i] == 'O')
              && (line[i+1] == 'r' || line[i+1] == 'R'))
              {
              line[i] = OR;
              line[i+1] = ' ';
              }
          }
        }
      else
        {
        printf("*** Error *** OR limit reached ");
        printf("*** iod parser0);
        printf("%s : %s0, iod-> header.ftype,
                iod-> header.name);
        }
      }
else
if (strcmp(token,"units") == 0)
```

```
{
/* parse units relation */
dwfp = fopen("DOUBLE.WORDS", "r");

/* skip white space */
while (isspace(line[++i]) != 0
      && line[i] != ' ');

if (line[i] == OR)
  {
  /* set up or index */
  uts_and_index = 0;
  if (uts_or_index == -1)
    uts_or_index++;
  if (strcmp(iod->units[uts_or_index]
          [uts_and_index].term, "") != 0)
    uts_or_index++;
  if (uts_or_index < FRAME_OR_MAX)
    strcpy(iod->units[uts_or_index+1]
            [uts_and_index].term, "");
  }

/* parse relation/attribute value */
if (uts_or_index < FRAME_OR_MAX)
  {
  while (line[i] != ' ')
    {
```

```
while (isalpha(line[++i]) == 0
        && line[i] != ' ');

if (isalpha(line[i]) != 0)
    {
    j = 0;
    if (isupper(line[i]) == 0)
      token[j] = line[i];
    else
      token[j] = tolower(line[i]);
    while (isalpha(line[++i]) != 0
         && line[i] != ' ')
     if (isupper(line[i]) == 0)
       token[++j] = line[i];
     else
       token[++j] = tolower(line[i]);
    token[++j] = ' ';

    if (strcmp(token, "and") != 0)
      {
      if (uts_and_index < FRAME_AND_MAX)
        {
        match = 0;
        if (uts_and_index > 0)
            {
            fseek(dwfp,0L,0);
            while (fgets(double_word,
```

```
                LINE_SIZE,dwfp)
                != NULL && match == 0)
        {
        sscanf(double_word,"%s %s",word1,
                        word2);
        if (strcmp(token,word2) == 0)
          {
          if (strcmp(iod-> units[uts_or_index]
                [uts_and_index-1].term,
                        word1) == 0)
              {
              strcat(iod-> units[uts_or_index]
                [uts_and_index-1].term," ");
              strcat(iod-> units[uts_or_index]
                      [uts_and_index-1].term,
                              token);
              match++;
              }
          }
        }
      }
if (match == 0)
    {
    strcpy(iod-> units[uts_or_index]
            [uts_and_index].term, token);
    uts_and_index++;
    strcpy(iod-> units[uts_or_index]
```

```
                    [uts_and_index].term, "");
                }
            }
            else
            {
            printf("0** ERROR *** AND limit reached0);
            printf(" *** iod parser0);
            printf("%s : %s0, iod-> header.ftype,
                    iod->header.name);
            printf("%s", line);
            }
          }
        }
      }
    else
      {
      printf("*** Error *** OR limit reached");
      printf(" *** iod parser0);
      printf("%s : %s0, iod-> header.ftype,
                iod->header.name);
      }
    fclose(dwfp);
    }
else
    {
    /* relation/attribute has been added to the */
```

```
        /* preprocessor but not to the parser or    */
        /* input, output, or uses was found in an   */
        /* iod relation.                            */
        printf("*** Warning *** unknown relation/attribute");
        printf("*** line flushed0);
        printf("%s",line);
        }
    }
  }
}
```

Checker

```
#include <stdio.h>
#include <ctype.h>
#include "era.def"
#include "era.struct"
#include "era.resolve.c"
#include "era.compare.c"
#include "era.error.c"

main(argc.argv)
int argc;
char *argv[];

{
int ffd; /* Function Data file descriptor */

struct fframe funct;
struct hframe header_frame;

char report_type[WORD_SIZE];

strcpy(report_type,*++argv);
printf("report type = %s0, report_type);

ffd = open("FUNCT",0);
if (ffd < 0)
```

```
    {
    printf("*** Error *** cannot");
    printf(" open file FUNCT0);
    }
else
  {
  lseek(ffd,0L,0);
  while (read(ffd,&funct,sizeof(funct)) != 0)
    {
    /* resolve function frame relations */
    resolve_relations(&funct,&header_frame);
    compare_units(&funct,&header_frame);
    print_errors(report_type);
    }
  close(ffd);
  }
unlink("FUNCT");
unlink("INDATA");
unlink("OTDATA");
unlink("TDATA");
unlink("UDATA");
}
```

Resolve Relations

```
#include <stdio.h>
#include <ctype.h>
#include "era.def"
#include "era.det.c"
#include "era.and.c"

resolve_relations(funct,header_frame)
struct fframe *funct;
struct hframe *header_frame;

{
int infd, /* Input Data file descriptor  */
    otfd, /* Output Data file descriptor */
    tfd,  /* Type Data file descriptor    */
    ufd;  /* Uses Data file descriptor    */

int i,
    j;

struct and_set *units_ptr;

infd = open("INDATA",0);
otfd = open("OTDATA",0);
ufd = open("UDATA",0);
tfd = open("TDATA",0);
```

```
if (infd < 0 || otfd < 0 ||
                    ufd < 0 || tfd < 0)
  {
  printf("*** Error *** cannot ");
  printf("open data files0);
  }
else
  {
  i = -1;
  j = 0;
  while (strcmp(funct->
          input[++i][j].term,"") != 0)
   {
   header_frame-> input[i] =
   (struct and_set *)
   determine_units(infd,funct->
                    input[i][j].term,tfd);
   while (strcmp(funct->
           input[i][++j].term,"") != 0)
    {
    units_ptr =
    (struct and_set *)
    determine_units(infd,funct->
                    input[i][j].term,tfd);
    and_units(header_frame->
                input[i], units_ptr);
    }
```

```
    j = 0;
  }
header_frame->input[i] = NULL;

i = -1;
j = 0;
while (strcmp(funct->
        output[++i][j].term,"") != 0)
  {
  header_frame->output[i] =
  (struct and_set *)
  determine_units(otfd,funct->
                output[i][j].term,tfd);
  while (strcmp(funct->
          output[i][++j].term,"") != 0)
    {
    units_ptr =
    (struct and_set *)
    determine_units(otfd,funct->
                output[i][j].term,tfd);
    and_units(header_frame->
                output[i],units_ptr);
    }
  j = 0;
  }
header_frame->output[i] = NULL;
```

```
i = -1;
j = 0;
while (strcmp(funct->
        uses[++i][j].term,"") != 0)
 {
 header_frame-> uses[i] =
 (struct and_set *)
 determine_units(ufd,funct->
            uses[i][j].term,tfd);
 while (strcmp(funct->
          uses[i][++j].term,"") != 0)
  {
  units_ptr =
  (struct and_set *)
  determine_units(ufd,funct->
                  uses[i][j].term,tfd);
  and_units(header_frame->
                    uses[i],units_ptr);
  }
 j = 0;
 }
header_frame->uses[i] = NULL;

close(infd);
close(otfd);
close(ufd);
close(tfd);
```

```
    }
}
```

Compare Units

```
#include <stdio.h>
#include <ctype.h>
#include "era.def"
#include "era.find.c"

compare_units(funct,header_frame)
struct fframe *funct;
struct hframe *header_frame;

{
int input_or_index,
   output_or_index,
   uses_or_index;

int input_and_index,
   output_and_index;

int match;

int no_error,
   uts_flag;

struct and_set *output_ptr,
               *input_ptr,
               *uses_ptr;
```

```
struct rel_val
      unresolved_units[AND_SET_MAX];

int unres_uts_index;

FILE *fopen(),
    *efp,
    *sfp,
    *cfp;

char line[LINE_SIZE],
    word[WORD_SIZE],
    syn[WORD_SIZE],
    con[WORD_SIZE];

efp = fopen("ERROR.REPORT", "a");

fputs(funct-> header.ftype, efp);
fputs(" : ", efp);
fputs(funct-> header.name, efp);
fputs("0, efp);

output_or_index = -1;
while (header_frame->output
      [++output_or_index] != NULL)
 {
 fputs("Processing output set -0, efp);
```

```
fputs("    ", efp);


output_and_index = -1;
while (strcmp(funct-> output[output_or_index]
        [++output_and_index].term,"") != 0)
 {
 fputs(" ", efp);
 fputs(funct-> output[output_or_index]
        [output_and_index].term, efp);
 }
fputs("0, efp);


/* process output set */
output_ptr = header_frame->
            output[output_or_index];
while (output_ptr != NULL)
 {
 fputs("    ", efp);
 output_and_index = -1;
 while (strcmp(output_ptr-> units
        [++output_and_index].term, "") != 0)
  {
  fputs(" ", efp);
  fputs(output_ptr-> units
        [output_and_index].term ,efp);
  }
 fputs("0, efp);
```

```
no_error = 0;
if (header_frame->input[0] == NULL)
  {
  uts_flag = 0;
  while (strcmp(output_ptr->units
        [++output_and_index].term,"") != 0)
      if (strcmp(output_ptr->units
        [output_and_index].term,"no match") != 0
        && strcmp(output_ptr->units
        [output_and_index].term,"no units") != 0)
       uts_flag++;
  if (uts_flag == 0)
      no_error++;
  }
input_or_index = -1;
while (header_frame->input
      [++input_or_index] != NULL)
 {
 fputs("Processing input set -0, efp);
 fputs("   ", efp);
 input_and_index = -1;
 while (strcmp(funct->input[input_or_index]
       [++input_and_index].term,"") != 0)
    {
    fputs(" ", efp);
    fputs(funct->input[input_or_index]
       [input_and_index].term,efp);
```

```
    }
fputs("0, efp);

/* process input set */
input_ptr = header_frame->
            input[input_or_index];
while (input_ptr != NULL)
    {
    fputs("    ", efp);
    input_and_index = -1;
    while (strcmp(input_ptr->units
        [++input_and_index].term, "") != 0)
     {
     fputs(" ", efp);
     fputs(input_ptr-> units
          [input_and_index].term, efp);
     }
    fputs("0, efp);

    strcpy(unresolved_units[0].term,"");
    unres_uts_index = -1;
    output_and_index = -1;
    while (strcmp(output_ptr->units
        [++output_and_index].term,"") != 0)
     {
     if (strcmp(output_ptr-> units
              [output_and_index].term,
```

```
        "no units") != 0 &&
 strcmp(output_ptr-> units
        [output_and_index].term,
        "no match") != 0)
{
match = find_unit(output_ptr->
         units[output_and_index].term,
         input_ptr);
if (match == 0)
    {
    /* check uses */
    uses_or_index = -1;
    while (header_frame->uses
          [++uses_or_index] != NULL)
     {
     uses_ptr = header_frame->
                 uses[uses_or_index];
     while (uses_ptr != NULL)
      {
      match = find_unit(output_ptr->
        units[output_and_index].term,
        uses_ptr);
      if (match != 0)
        uses_ptr = NULL;
      else
        uses_ptr = uses_ptr->or_set;
      }
```

```
        }
      }
if (match == 0)
      {
      /* check synonyms */
      sfp = fopen("SYNONYM.DATA","r");
      while (fgets(line, LINE_SIZE,
            sfp) != NULL && match == 0)
       {
       sscanf(line, "%s : %s", word, syn);
       if (strcmp(output_ptr->units
          [output_and_index].term,word) == 0)
         {
         match = find_unit(syn,input_ptr);
         if (match == 0)
             {
             /* check for synonym in */
             /* uses structure       */
             uses_or_index = -1;
             while (header_frame->uses
                 [++uses_or_index] != NULL)
              {
              uses_ptr = header_frame->
                         uses[uses_or_index];
              while (uses_ptr != NULL)
               {
               match = find_unit(syn,uses_ptr);
```

```
                if (match != 0)
                  uses_ptr = NULL;
                else
                  uses_ptr = uses_ptr->or_set;
                }
              }
            }
          }
        }
      fclose(sfp);
      }
  if (match == 0)
      {
      /* check conversions */
      cfp = fopen("CONVERSION.DATA","r");
      while (fgets(line, LINE_SIZE,
           cfp) != NULL && match == 0)
       {
       sscanf(line, "%s : %s", word, con);
       if (strcmp(output_ptr->units
          [output_and_index].term,word) == 0)
         {
         match = find_unit(con,input_ptr);
         if (match == 0)
             {
             /* check for conversion in */
             /* uses structure          */
```

```
            uses_or_index = -1;
            while (header_frame->uses
                [++uses_or_index] != NULL)
              {
              uses_ptr = header_frame->
                  uses[uses_or_index];
              while (uses_ptr != NULL)
                {
                match = find_unit(con,uses_ptr);
                if (match != 0)
                  uses_ptr = NULL;
                else
                  uses_ptr = uses_ptr->or_set;
                }
              }
            }
        }
    }
fclose(cfp);
if (match != 0)
  {
  /* print conversion warning */
  fputs("  Warning - conversion required ",
                          efp);
  fputs(output_ptr->units
        [output_and_index].term, efp);
  fputs("0, efp);
```

```
        }
      }
  if (match == 0)
      {
      strcpy(unresolved_units[++unres_uts_index].term,
          output_ptr->units[output_and_index].term);
      strcpy(unresolved_units[unres_uts_index+1].term,
                                    "");
      }
  }
}
unres_uts_index = 0;
if (strcmp(unresolved_units
   [unres_uts_index].term, "") != 0)
  {
  /* print warning */
  fputs("  Warning - unresolved units -", efp);
  while (strcmp(unresolved_units
         [unres_uts_index].term, "") != 0)
         {
         fputs(" ", efp);
         fputs(unresolved_units
           [unres_uts_index++].term, efp);
         }
  fputs("0, efp);
  }
else
```

```
            no_error++;
          input_ptr = input_ptr->or_set;
          }
      }
    if (no_error == 0)
      {
      fputs("Error - output units set was", efp);
      fputs(" never produced0, efp);
      }
    output_ptr = output_ptr->or_set;
    }
  }
fclose(efp);
}
```

Print Errors

```
#include <stdio.h>
#include <ctype.h>
#include "era.def"

print_errors(report_type)
char report_type[];

{
FILE *fopen(),
    *efp;

int match;

char line[LINE_SIZE],
    word[WORD_SIZE];

efp = fopen("ERROR.REPORT", "r");

match = 0;
if (strcmp(report_type,"short") == 0)
  {
  while(fgets(line, LINE_SIZE,
            efp) != NULL && match == 0)
        {
        sscanf(line, "%s", word);
```

```
        if (strcmp(word,"Warning") == 0)
           match++;
        }
    }
else
   match++;

if (match != 0)
   {
   fseek(efp,0L,0);
   while(fgets(line, LINE_SIZE, efp) != NULL )
         printf("%s",line);
   }

fclose(efp);
unlink("ERROR.REPORT");
}
```

Determine Units

```
#include <stdio.h>
#include <ctype.h>
#include "era.def"

struct and_set *
determine_units(fd,word,tfd)
int fd;
char word[];
int tfd;

{
int match;

int i,
   j;

struct iframe iod;
struct and_set *and_ptr,
               *or_ptr,
               *units_ptr;

char *malloc();

lseek(fd,0L,0);
match = 0;
```

```
while (read(fd,&iod,sizeof(iod)) != 0
        && match == 0)
 {
 if (strcmp(word,iod.header.name) == 0)
   {
   /* match found */
   match++;

   if (strcmp(iod.units[0][0].term,"") != 0)
       {
       /* units found - copy units */
       i = -1;
       j = 0;
       units_ptr = (struct and_set *)
           malloc(sizeof(struct and_set));
       if (units_ptr == NULL)
         {
         printf("*** Error *** malloc ***");
         printf(" units pointer error0);
         }
       and_ptr = units_ptr;
       while (strcmp(iod.units
                [++i][j].term,"") != 0)
        {
        strcpy(and_ptr->units[j].term,
                  iod.units[i][j].term);
        while (strcmp(iod.units
```

```
                [i][++j].term,"") != 0)
            strcpy(and_ptr-> units[j].term,
                iod.units[i][j].term);
        strcpy(and_ptr->units[j].term,"");

        j = 0;

        /* look ahead for more units */
        if (strcmp(iod.units[i+1][j].term,"") != 0)
          {
          or_ptr = (struct and_set *)
              malloc(sizeof(struct and_set));
          if (or_ptr = NULL)
              {
              printf("*** Error *** malloc");
              printf(" *** or pointer error0);
              }
          and_ptr-> or_set = or_ptr;
          and_ptr = or_ptr;
          }
        else
          and_ptr-> or_set = NULL;

      }
    }
else
    {
```

```
/* no units - process structure */
/* and type                     */
if (strcmp(iod.type[0][0].term,"") != 0)
  {
  /* process type */
  i = -1;
  j = 0;
  units_ptr = NULL;
  while (strcmp(iod.type
            [++i][j].term,"") != 0)
   {
   or_ptr = (struct and_set *)
            determine_units(tfd,iod.type
                        [i][j].term,tfd);
   if (units_ptr == NULL)
       units_ptr = or_ptr;
   while (strcmp(iod.type[i][++j].
                  term,"") != 0)
    {
    and_ptr = (struct and_set *)
          determine_units(tfd,iod.type
                        [i][j].term,tfd);
    and_units(or_ptr,and_ptr);
    }

   /* connect or set */
   if (units_ptr != or_ptr)
```

```
        {
        and_ptr = units_ptr;
        while (and_ptr->or_set != NULL)
          and_ptr = and_ptr->or_set;
        and_ptr->or_set = or_ptr;
        }
      j = 0;
      }
    }
  else
    if (strcmp(iod.structure
            [0][0].term,"") != 0)
      {
      /* process structure */
      i = -1;
      j = 0;
      units_ptr = NULL;
      while (strcmp(iod.structure
            [++i][j].term,"") != 0)
        {
        or_ptr = (struct and_set *)
          determine_units(tfd,iod.
          structure[i][j].term.tfd);
        if (units_ptr == NULL)
          units_ptr = or_ptr;
        while (strcmp(iod.structure
            [i][++j].term,"") != 0)
```

```
        {
        and_ptr = (struct and_set *)
            determine_units(tfd,iod.
            structure[i][j].term.tfd);
        and_units(or_ptr,and_ptr);
        }

      /* connect or set */
      if (units_ptr != or_ptr)
        {
        and_ptr = units_ptr;
        while (and_ptr-> or_set != NULL)
            and_ptr = and_ptr-> or_set;
        and_ptr-> or_set = or_ptr;
        }
      j = 0;
      }
  }
else
  {
  /* no units found */
  units_ptr = (struct and_set *) malloc
              (sizeof (struct and_set));
  if (units_ptr == NULL)
      {
      printf("*** Error *** malloc");
      printf(" *** no units pointer error0);
```

```
                }
            units_ptr->or_set = NULL;
            strcpy(units_ptr-> units[0].term,"no units");
            strcpy(units_ptr-> units[1].term,"");
            }
        }
    }
  }
if (match == 0)
  {
   units_ptr = (struct and_set *) malloc
                (sizeof(struct and_set));
   if (units_ptr == NULL)
     {
     printf("*** Error *** malloc ***");
     printf(" no match pointer error0);
     }
   units_ptr-> or_set = NULL;
   strcpy(units_ptr-> units[0].term,"no match");
   strcpy(units_ptr-> units[1].term,"");
   }
return(units_ptr);
}
```

Find Units

```
#include <stdio.h>
#include <ctype.h>
#include "era.def"

int
find_unit(unit,and_ptr)
char unit[];
struct and_set *and_ptr;

{
int i;
int match;

i = -1;
match = 0;

while ((strcmp(and_ptr-> units[++i].term,"")
            != 0) && match == 0)
  if (strcmp(unit,and_ptr->
            units[i].term) == 0)
    match++;
return(match);
}
```

And Units

```
#include <stdio.h>
#include <ctype.h>
#include "era.def"

and_units(or_ptr,and_ptr)
struct and_set *or_ptr;
struct and_set *and_ptr;

{
int i,
    j;

int count;

char *malloc();

struct and_set *units_ptr,
               *cp_ptr,
               *next_ptr;

if (and_ptr != NULL)
  {
  /* count nodes */
  units_ptr = and_ptr->or_set;
  count = 1;
```

```
while (units_ptr != NULL)
 {
 units_ptr = units_ptr-> or_set;
 count++;
 }

/* copy nodes */
units_ptr = or_ptr;
while (units_ptr != NULL)
 {
 next_ptr = units_ptr-> or_set;
 for (i = 1; i < count; i++)
  {
  cp_ptr = (struct and_set *) malloc
            (sizeof(struct and_set));
  if (cp_ptr == NULL)
      {
      printf("*** Error *** malloc ***");
      printf(" and units pointer error0);
      }
  units_ptr-> or_set = cp_ptr;
  j = -1;
  while (strcmp(units_ptr->
            units[++j].term,"") != 0)
      strcpy(cp_ptr-> units[j].term,
            units_ptr-> units[j].term);
  strcpy(cp_ptr-> units[j].term,"");
```

```
  units_ptr = cp_ptr;
  }
units_ptr-> or_set = next_ptr;
units_ptr = next_ptr;
}
units_ptr = or_ptr;
while (units_ptr != NULL)
 {
 /* and nodes */
 cp_ptr = and_ptr;
 while (cp_ptr != NULL)
    {
    i = -1;
    while (strcmp(units_ptr->
          units[++i].term,"") != 0);
    j = -1;
    while (i < AND_SET_MAX && (strcmp
         (cp_ptr->units[++j].term,"") != 0))
      {
      strcpy(units_ptr-> units[i].term,
                  cp_ptr-> units[j].term);
      i++;
      }
    if (strcmp(cp_ptr-> units[j].term,"") == 0)
      strcpy(units_ptr-> units[i].term,"");
    else
      {
```

```
            printf("*** Error *** too many units");
            printf(" *** and_units0);
            strcpy(units_ptr-> units[i].term,"");
            }
          cp_ptr = cp_ptr-> or_set;
          units_ptr = units_ptr-> or_set;
          }
      }
    }
}
```

THE UNITS OF MEASURE CONSISTENCY CHECKER FOR THE ENTITY-RELATIONSHIP-ATTRIBUTE REQUIREMENTS MODEL

by

GALE LYNN METZ

B. S., University of Illinois, Urbana, 1977

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

Kansas State University
Manhattan, Kansas

1986

Abstract

As software systems become larger and more complex, many man hours can be wasted designing and testing software that has been developed from incorrect specifications. Therefore, increasing effort is being devoted to uncovering errors as early as possible in the development cycle. Much of this effort is being directed at formalizing the language used in the requirements specification document. This formalization will eventually allow computer aids to help ensure the correctness of the requirements specification document. One such formalism is the Entity-Relationship-Attribute (E-R-A) Requirements Specification language, which uses a frame oriented data structure to represent real-world knowledge. This type of representation allows the specification to be checked for various forms of correctness.

This report describes a units of measure consistency checker that has been developed for the E-R-A model. It is part of a group implementation project. This portion of the system will verify that the specified output units of measure can be obtained from the given input units of measure. In order to accomplish this goal, it first parses the E-R-A Specification, searches until all the input and output units of measure have been obtained and then verifies that the necessary units of measure exist to obtain the required output units of measure.