THE IMPLEMENTATION OF AN INPUT/OUTPUT CONSISTENCY CHECKER FOR A REQUIREMENTS SPECIFICATION DOCUMENT

by

LAURA HAZEL WELMERS

B.S., Michigan State University, 1977

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

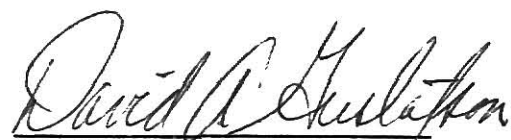
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

Approved by:



Dr. David A. Gustafson

Table of Contents

Chapter One

Consistency Checking of a Requirements Specification Document

1.1 Introduction

This report describes a tool for checking consistency within a requirements specification document. The input to this tool is a requirements specification document. The consistency checker reads the document and extracts and compares keyword related information. The resulting output is a report listing the various inconsistencies encountered within the requirements specification document.

The software development life cycle is composed of a number of phases. The set of phases, requirements analysis, requirements specification, preliminary design, detailed design, coding, testing, and operations and maintenance, is one way to break down the life cycle of a software development project [Pr82]. Each of these phases has a distinct goal to achieve. The sum of all of these phases will together produce the desired product of the software development project.

The requirements specification phase is one of the most important phases of the software life cycle. Once an analysis has been done on the requirements of a project, and the project is determined to be feasible, the specifications must be generated.

The requirements specification document defines the acceptable implementation of a system, along with any imposed constraints [He83]. It is extremely important that the requirements be specified in an adequate, consistent and complete manner. All of the phases after the requirements specification will depend on what is generated from the specification phase. The better the requirements specifications are defined, the smoother the rest of the life cycle phases should proceed.

There are numerous methods for developing requirements specification documents. The more popular methods range from SADT (Structured Analyses and Design Technique) to PSL/PSA (Problem Statement Language/Problem Statement Analyzer) to SREM (Software Requirement Engineering Methodology) [Pr82]. In the SADT method, a graphical box and arrow notation is used to describe both data decomposition and activity decomposition in a hierarchic model [Ro77]. PSL/PSA uses a relational data base to identify and name objects and their relationships from the language statements; and provides an analyzer to generate reports [Te77]. The SREM methodology expresses software requirements using a flow-oriented language, and then uses a translator for that language with a data base to maintain the information derived from the requirements [Be77]. Requirements modeling framework (RMF) is another method for developing requirement specification documents. RMF incorporates three concepts, entity categories, property categories and abstraction mechanisms [Gr82]. Requirement specification languages can encompass the wide variety of existing methods.

Entity-Relationship-Attribute (e-r-a) is one form a requirements specification document may take. The e-r-a specification is created by defining entities, describing relationships between entities and providing attributes about entities. There are some similarities between the format of the e-r-a specification and the RMF method as well as in the entity and relationship concept. The e-r-a specification provides a standard, specific format for generating a requirements document.

A requirements specification document should be in a correct and complete form, as early on as possible. Since the remainder of the software life cycle phases depend on the completion of the requirements specification phase, the sooner that phase is completed the sooner succeeding portions of the project can get underway. The length of time required to generate a requirements specification document is not the only important aspect. The document should be correct and complete as well. Incomplete or incorrect specifications will only lead to problems in the later phases of the life cycle.

Checking for consistency in the requirements specification document assists in ensuring its validity. Inconsistencies in the specification document, left unheeded, can lead to complications in later phases. Checking for consistency in the early stages helps to reduce the number of errors found in later stages. The implementation of a consistency checker will help increase the likelihood that a requirements specification document is valid. If a specification document is consistent within itself, fewer

problems should be encountered later.

The development of software tools can help improve the usefulness of the requirements specification document. Software tools can provide an efficient and flexible means to generate various software related items [Re79]. Software tools can provide an automated way to perform various tasks. Automation generally will allow the user to perform in a quicker manner and at the same time with more accurate results [Ye83]. Tools that allow the user to generate an efficient, accurate requirements specification document can only enhance the development process.

Automation of software tools, such as a consistency checker, is part of the 5th generation software engineering field [Tr82]. As the cost of software rises, in relation to the total cost of a computer based software system, tools become more important in the software development area [Re79]. Consistency checking, whether done automatically or manually, is a necessary task when creating a requirements specification document. Automated tools provide a faster method to achieve the end goal, in this case a consistent requirements specification document.

1.2 Consistency Checking

Completeness, traceability and comprehensibility are issues that are discussed in conjunction with consistency checking. These are all desirable properties of a specification document [Wa81]. Traceability of items, within the specification document, helps

ensure its consistency. The act of checking consistency, along with completeness, are two of the criteria used for verification and validation of the requirements specification document [Bo84]. Verification is the process of determining whether the products of some phase in the software life cycle satisfy the requirements of the previous phase [Bo84]. Validation is the process of evaluating the final software product to determine whether the product meets the software requirements [Bo84].

Quality assurance is one of the main focal points in the software area. A major goal of software quality assurance is consistency. Achievement of consistency in the requirements specification phase is important because future phases could be affected by errors that could be propagated by inconsistencies not detected. Finding errors or inconsistencies in the early stage of the life cycle can reduce the cost of the project by eliminating errors in later stages [Ny83]. A key to the successful development of a reliable software system is verifying that a requirements specification is correct [Ag82].

Software projects range in size from the very small to the very large. The tasks of verification and validation of software project specifications, in essence remain the same regardless of the size of the project. Methods of verifying the specification that can be done by hand for small projects must become automated for large projects. The amount of checking required increases faster than the size of the software project being developed.

With large software projects come an increase in the number of people (hence inter-communications) working together. When there are a number of people working on different parts of a project, the ability to maintain consistency becomes very difficult [Ti81]. With different implementors working on different interfaces, misunderstandings become more a rule than an exception [Ti81]. Automated software tools can be used to eliminate some of the bad side-effects of increased inter-communications between project members.

Increased modularity aids in developing the ever larger software projects appearing in today's world. Breaking software projects into modules helps the programmer/analyst better understand each portion of the project. Specifications are generally better understood on a small scale. With programs broken down into modules, though, comes the task of combining them all together to form the total picture.

Controlling the interfaces between modules can help to maintain consistency. With software projects broken down into small modules, the number of interfaces between modules will increase. With increased interfaces, the likelihood of errors occurring will generally increase. Exerting some control over the interfaces and inter-connections can help maintain a desirable level of consistency.

The language or procedure used to develop a requirements specification document helps to determine its ability to work with

software tools. Certain types of languages are better suited to work with automated tools than others. A formal language should have a processor that performs syntactic and semantic checking to ensure completeness, consistency, unambiguity and testability [Da79]. Most requirement specifications are written in ambiguous languages. To entirely automate the verification process, specifications would have to be written in an unambiguous grammar of a previously defined language [Be76]. This is not the practical approach, though, because non-computer oriented people, with different backgrounds, must be able to interpret the requirements [Da79]. Some structure must exist in the language used to develop the requirements specification, though, to make the resulting document usable. The ideal requirements language would be formal as well as human-engineered for customers [Da79].

There are more advantages in obtaining a consistent requirements specification document. The remainder of the phases of the life cycle are dependent on the requirements specification. Generating a valid requirements specification helps ensure that the design and coding phases are geared to achieve the final goal. A consistent requirements specification helps lead to a valid specification. A valid specification is one that can be verified. Specification verification is a key to improving the reliability of software [Ag82]. A major time saving step, in the life of a large computer-based software system, is to have completely verified specifications prior to the actual development of the software [Be76]. Many hours can be wasted in later life-cycle stages if

erroneous specifications are used.

1.3 The Consistency Checker

Consistency checking of an e-r-a specification document can be performed in a number of ways. This implementation will check the consistency of inputs and outputs related to "Activity" and "Periodic Function" entities. It will check that all inputs and outputs are both defined and used. Other checking that could be done involves checking unit types for consistency. Data flow, derived from the requirements specification, is another check that could be performed by comparing functions and data definitions. Traceability of items in the specification, (those that have defined antecedents in previous specifications or object items), is an additional consistency checking manipulation [Bo84]. Incorporating just one checking procedure can aid in validating the requirements specification document.

In addition to consistency checking in the requirements specification phase, work has been done at the design level phase of the life cycle. TRW observed, with an analysis of data, that cost-effectiveness could be achieved with the introduction of an automated tool to detect inconsistencies in the design phase [Bo75]. GTE found a way to handle the inconsistency problem, between design and implementation phases, by introducing an embedded design language (EDL) with an associated consistency checker [Ru82]. To incorporate a design language, the language to

be used for the implementation should be considered. Strong typing languages, such as Pascal, already provide a consistency checking feature through their compiler. Building extensions on such a language was the method GTE used to create their EDL [Ru82].

Testing comprises a major portion of the life cycle. The standard "testing" phase of the life cycle occurs after the "coding" phase. This "testing" phase occurs directly before the software product is placed into production. A great deal of effort is exerted in this phase of the life cycle to ensure that the final product meets the specifications outlined in the requirements specification document. Some types of testing can be done within all phases of the life cycle as well as in the standard "testing" phase. Consistency checking, in the requirements specification phase, is a way of testing the specifications and could reduce some of the testing needed in later phases. The more correct the requirements specification document is, the fewer the number of errors there should be later on, which in itself should reduce the amount of testing required.

The ability to maintain consistency with change would be ideal [We82]. Obtaining a consistent requirements specification document is a step in the right direction. Perhaps more important, though, is to maintain consistency when requirements change. Most computer based software projects are not static. Often specifications are not designed properly to be flexible with change. The advent of automated tools can possibly alleviate introducing errors when changes are made. Automated tools could allow the user to

automatically invoke changes on their requirements specification document, in this case, whenever there is a change made to the requirements. This will then allow consistency to be maintained while there is change.

Formality between the different phases of the life cycle could enhance consistency within all levels. The more formal the method used within the individual phases in the life cycle, the more likely it is that consistency will be maintained at each phase. Formal representations of, and uniform interfaces between the different phases can provide a means to check consistency [Ca81]. The later phases of the life cycle are generally more formal because they use a more rigorous language.

1.4 Summary

Ensuring a good requirements specification document is becoming a goal of the software engineering community. History has shown that there is a natural breakdown into the different phases of the software life cycle. These different phases compare to the different types of activities that must be performed to develop a software project. The requirements specification phase is one of the early phases of the software life cycle. The product of the requirements phase is the requirements specification document. Since this document is produced in an early phase of the total software development project it is imperative that it be as complete and accurate as possible. By ensuring a good document at

this phase, succeeding phases should be easier to produce.

Providing software tools, to be used on the requirements specification document, helps to ensure the document's usefulness throughout the software life cycle. Automated software tools allow the implementor to perform tasks easier and generally more efficiently. The users of software tools can generate a product that is less prone to errors since automated tools should perform in the same manner all the time. A requirements specification document that is created using software tools will be instrumental in deriving good products in the later life cycle phases.

Consistency checking is one step in generating a valid requirements specification document. As an automated tool, a consistency checker of the requirements specification document can inspect the document and detect inconsistencies. Checking for consistency is one of the first steps that can be performed on a requirements specification document. A consistency checker is an ideal tool to use after every iteration of the specification document. Achieving consistency, in a requirements specification document, is one major step to generating a valid document to be used as the basis for the development of the desired software product.

Chapter Two

Requirements and Design Decisions

2.1 Requirements

A requirements specification document should be an accurate reflection of what is being described. Errors, left uncorrected in the requirements specification phase, can propagate to later phases of the software development life cycle. Errors in the requirements specifications, if unfound, can be expensive to correct during later phases. The importance of a valid, correct, complete and consistent requirements specification document *can not* be over stated.

Checking for consistency in a requirements specification document is a means to ensure its validity. Before a requirements specification can be determined to be valid, it must be correct within itself. A step to obtaining a correct requirements specification is to ensure consistency within the document itself. An automated software tool to check for consistency can assist the requirement specification analyst in generating a valid requirements specification document.

To write any kind of automated software tool, certain guidelines have to be defined. When creating automated tools, one would like to assume a predefined standard format for the input. Another standard would be to set up how the output is generated and

reported to the user. Sometimes the output from one software tool may be the input to another software tool. The more sophisticated software tools may even be interactive, in that the user is prompted for information and then the tool acts upon that information. All of these areas must be addressed when developing software tools.

This consistency checking tool allows the user to have an entity-relationship-attribute (e-r-a) requirements specification document interpreted for inconsistencies of input and output entities as used with activity and periodic function entities. In this implementation, consistency checking of an e-r-a specification document includes a number of aspects. The major feature of this tool is to identify any inputs or outputs, of an e-r-a requirements specification document, that are either used in "activity" or "periodic function" entities and not defined as input or output entities; and to identify any input or output entities that are not used in any "activity" or "periodic function" entities. In addition, the input and output entities are checked to ensure that they are fully defined by type or structure and that no "type" entities exist that aren't being used.

2.2 Data Flow

The consistency checking tool reads in an e-r-a requirements specification document and produces messages indicating any inconsistencies encountered as shown in Figure 1.

Data Flow Diagram

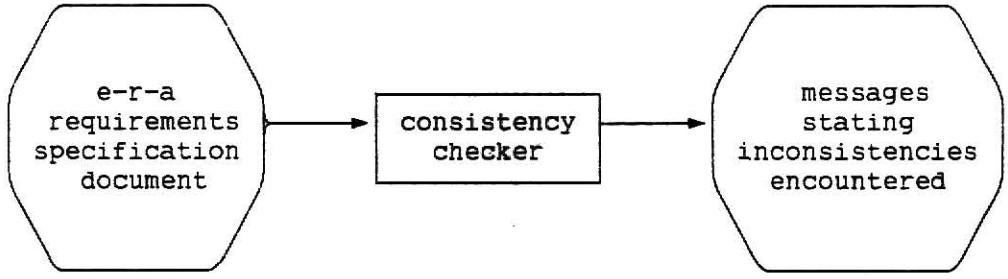


Figure 1.

The consistency checking tool can be depicted as performing two major functions. First the input is syntactically scanned and internal tables are created containing the entities and their corresponding attributes (see Figure 2). After the input file, containing the e-r-a requirement specifications, has been read in, consistency checks are performed and messages, depicting inconsistencies encountered, are generated on the terminal (or optionally to a file by redirecting the output).

Program Flow Diagram

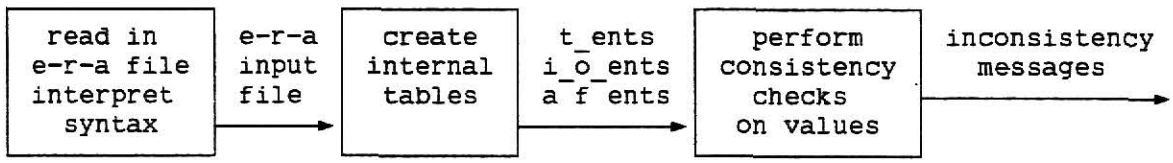


Figure 2.

2.3 Input

A text file containing the e-r-a requirements specification document is the required input. An e-r-a specification document, conforming to the defined Backus-Naur-Form (b-n-f) grammar (see Appendix A), is required to ensure a common format for all acceptable input. A software tool could generate an e-r-a requirements specification document, following the rules defined by the b-n-f grammar. Checking for typographical errors, for the most part, can be avoided by having another software tool create the input. The input text file will reside on a UNIX* operating system.

The e-r-a requirements specification document is divided into a defined set of entities, each with their own set of possible keyword attributes. The e-r-a specification document contains more entities and attributes than the consistency checker uses. To provide flexibility, the consistency checking tool will be table driven. All the possible entity names, that are used in conjunction with the consistency checker, will form the basis of the table along with each of their corresponding applicable attributes. If additions or deletions are made to the format of the e-r-a requirements specification document, any necessary changes should be able to be made to the table with a minimal

---

* UNIX is a Trademark of AT&T Bell Laboratories.

amount of effort.

2.4 Output

The output generated from the consistency checker is a list of error messages detailing the inconsistencies encountered. Depending on the type of error, a certain message will be reported, indicating the offending entity and its inconsistency. Errors include inputs or outputs used in activities or periodic functions that are not defined as input or output entities; or input or output entities not used in any activity or periodic function. Other errors include inputs or outputs not fully defined by structure type entities; or type entities not used in inputs or outputs. The user should then act upon those error messages and make the necessary corrections to the e-r-a requirements specification document.

2.5 Functions

This consistency checker implementation requires that the entire e-r-a requirements specification document be read and interpreted first. After the applicable information for each entity has been extracted, the different consistency checks can be performed. The 'Activity' and 'Periodic_function' entities are checked first against the inputs and outputs. The inputs and outputs are then checked to see that they have all been used. Those inputs and outputs that have been used are then checked to make sure that all of the structures related to them are defined as

types. Finally all types are checked to ensure that they are completely defined, as well as used in an input or output.

2.6 Implementation

Implementation was done on the Kansas State University Perkin-Elmer 8/32 computer using version 7 of the UNIX operating system. The C programming language was used to write the consistency checking program. The tool must be used in a UNIX-based environment, with a C language library facility.

Chapter Three

Design of the Input/Output Consistency Checker

An entity-relationship-attribute (e-r-a) requirements specification document is interpreted by the consistency checker tool and generates a report detailing the inconsistencies encountered. An input file containing an e-r-a document with entities and associated keyword attributes is read in and lexically scanned for certain keywords in a table driven format. Specific information about certain entities is retained. Different types of consistency checks are then performed on the retained information. A report is then generated detailing the inconsistencies uncovered in the e-r-a document.

A hierarchy diagram of the consistency checker program is displayed in Figures 3a, 3b and 3c. Figure 3a portrays the overall hierarchy of the consistency checking tool. The specification document is read in and the necessary information is extracted. The different consistency checks are then performed on the extracted information.

For debugging purposes there is a function print_tbls that will print the internal tables created after the entire e-r-a requirements specification document has been read in.

Hierarchy Overview

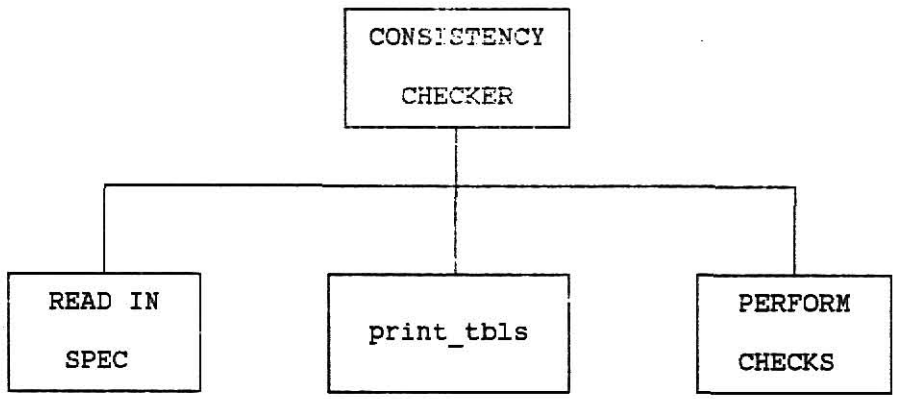


Figure 3a.

In figures 3a, 3b and 3c those boxes with entries printed in lower case correspond to the actual function names in the source code. Appendix C contains a module specification of the consistency checker in relation to the source code in Appendix D. Each functions purpose is depicted in the box(es) directly below each function name in the hierarchy diagrams. The Module Specification in Appendix C also gives a broader description of each function.

Figure 3b details interpreting the requirements specification document. Based on the type of entity encountered, certain attributes are extracted and related to that entity.

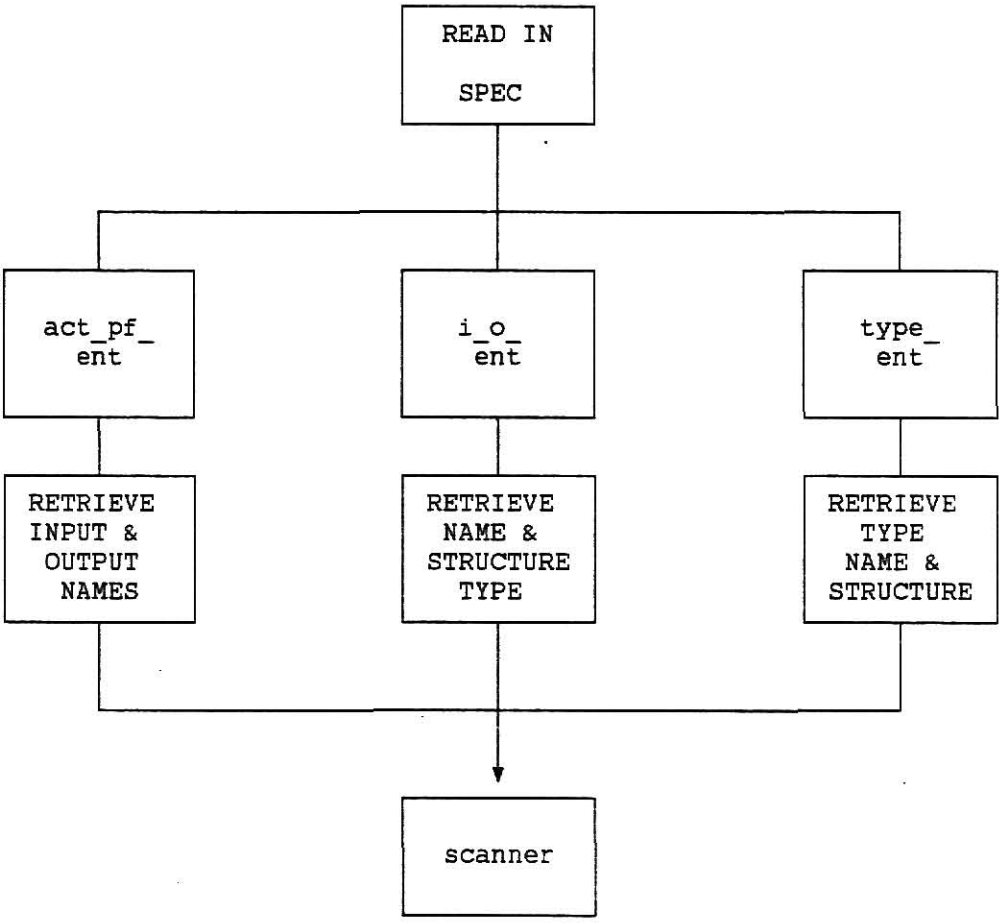


Figure 3b.

Figure 3c breaks down the different checks that are performed on the extracted information. Each checking routine generates the specific error message for the corresponding inconsistency.

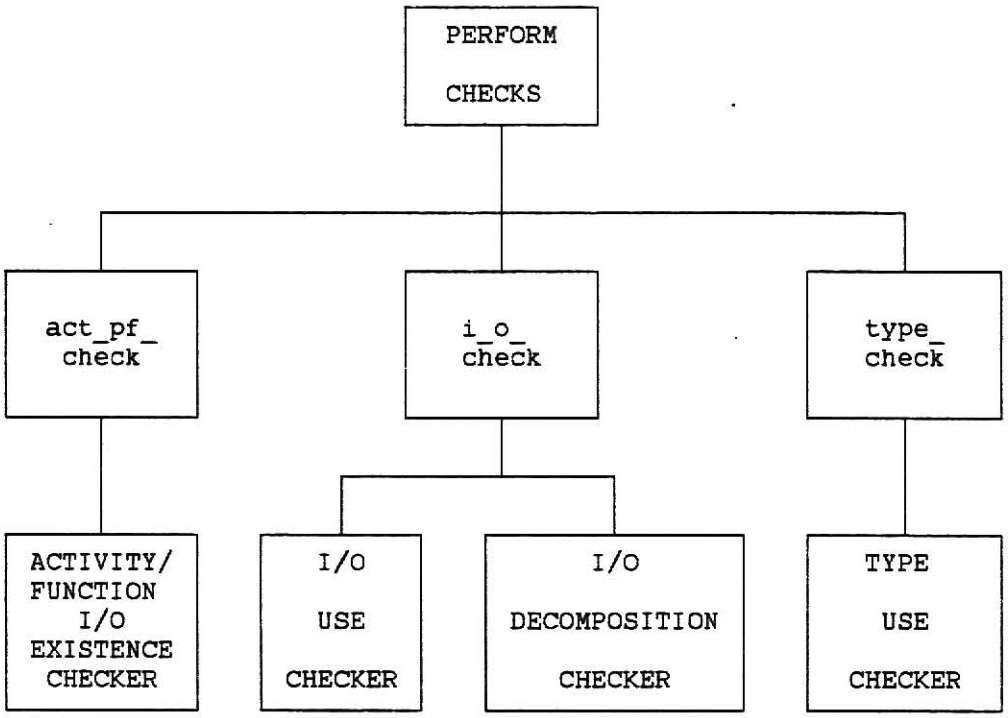


Figure 3c.

A sample template of an e-r-a requirements specification document is displayed in Figure 4. Not all the entities, nor their corresponding attributes, are shown in the example. For a full description of the e-r-a requirements specification document see the complete Backus-Naur-Form (b-n-f) grammar in Appendix A. The consistency checker only uses the Activity, Periodic_function, Input, Output, Input_output, Data and Type entities. The input, output and structure attributes are the only attributes the consistency checker uses.

Sample E-R-A Specification Format

```
Activity:  Name
   keywords:
   input:  $variable$
   output:  $variable$
   required-mode:
   necessary-condition:
   assertion:
   action:


Periodic_function:  Name
   required-mode:
   occurrence:
   output:  $variable$
   action:


Input:  $variable$
   media:
   structure:  'string' and/or $variable$
   contents:


Output:  $variable$
   media:
   structure:  'string' and/or $variable$
   contents:


Input_output:  $variable$
   media:
   structure:  'string' and/or $variable$
   contents:


Data:  $variable$
   structure:  'string' and/or $variable$
   contents:


Type:  $variable$
   structure:  'string' and/or $variable$ and/or set {...}
```

Figure 4.

For the "Activity" and "Periodic_function" entities, the inputs and outputs associated with each are recorded. The "Activity" and "Periodic_function" entity information is stored in a data structure that contains the name of the entity, the count of the number of inputs and outputs and two sub-structures containing the actual input and output names. When an "Activity" or "Periodic_function" entity is encountered, the name of the activity or function is recorded and then the keyword listed inputs and outputs are recorded. Multiple inputs and/or outputs can occur with each activity and function.

For the "Input", "Output", "Input_output" and "Data" entities, structure information is extracted. The "Input/Output" data structure contains the name of the entity, two flags to indicate whether the entity is an input, output or both, two use counts and a sub-structure to hold the possible type structures. For each of the "Input", "Output", "Input_output" and "Data" entities, the name of the entity and its corresponding structure is retained. The structure is examined and determined to be either a character string or a user defined type variable. User defined type variables are defined (and sometimes redefined) in "Type" entities.

For the "Type" entities, structure information is extracted. The "Type" data structure is similar to the "Input/Output" data structure. It contains the name of the entity, a use count and a sub-structure containing possible redefined type structures. When a "Type" entity is encountered, the type name and its corresponding structure are recorded. The structure will either be some

combination of character strings and/or additional user defined type variables.

The consistency checker then does cross-checking with the activities and periodic functions against the inputs and outputs. Each activity and function name is examined. All inputs and outputs associated with each is cross-checked against the input, output, input_output and data entity information. Each input and output item is counted each time it is referenced. The input_output and data items have flags to mark that it is an input and that it is an output. Any activity or function that references an undefined input or output generates an error message indicating the inconsistency. After all of the activities and functions are checked, a reverse check is done. The list of inputs and outputs are scanned and any that have a zero use count generates an error message indicating the inconsistency. Each input_output and data item is checked to insure it is flagged as both an input and an output, and generates an inconsistency error message when it is not.

Further checking of the inputs and outputs, to ensure they are completely defined, is also performed. Any input, output, input_output or data item that uses a user defined type variable in its structure is further checked. The user defined type variable is checked against the "Type" entities to be sure it is defined. If a user defined type variable is not defined in a type entity, an error message indicating the inconsistency is reported. Each user defined type variable that is referenced, is counted. All user

defined type entities that are further defined by another user defined type variable are checked until they are finally defined by a character string or set. Finally all "Type" user defined type variables are examined. Any user defined types not accessed, generate an inconsistency error message.

Inconsistencies are detailed in a generated report. All the different inconsistency error messages are recorded in the report. The format of the different error messages is shown in Figure 5.

Sample Error Messages

ACTIVITY _____ USES _____ WHICH IS NOT DEFINED
AS AN input OR data item.

ACTIVITY _____ PRODUCES _____ WHICH IS NOT DEFINED
AS AN output OR data item.

_____ NOT USED AS AN input TO ANY ACTIVITY

_____ NOT AN output FROM ANY ACTIVITY

_____ NOT AN input OR output TO/FROM ANY ACTIVITY

_____ NOT DEFINED AS A type

_____ type NOT USED AS AN input OR output

_____ type NOT DEFINED BY ANOTHER type

Figure 5.

Chapter Four

Implementation of the Input/Output Consistency Checker

Implementation of the input/output consistency checker, for a requirements specification document, was done on the KSU Perkin-Elmer 8/32 computer using the UNIX operating system. The C programming language was used to write the consistency checker. The expected input to the consistency checker is a UNIX file containing an entity-relationship-attribute (e-r-a) requirements specification document. The input is assumed to be directed from standard input. The output generated from the consistency checker is a UNIX file containing a report detailing the inconsistencies found in the requirements specification document. The output is expected to be directed from standard output.

The consistency checker program is composed of a number of subroutines, in addition to the main controlling program. A separate file contains a table structure of the entities and attributes, from the e-r-a specification, that are specifically used by the consistency checker program. If modifications are made to the entity or attribute names, that affect the consistency checker, only that file would need to be changed. Another file contains the structures of the internal tables used to store the information extracted from the requirements specification document. All of the static defined constants, used in the consistency checker program, reside in yet another file.

The subroutines that comprise the consistency checker program are functionally independent. Based on the entity encountered, a separate routine is called for an activity or periodic function entity; for an input or output entity (including input_output and data entities); and for a type entity. Each of those three routines calls a common scanning routine to extract the appropriate attribute information. Once the entire e-r-a document has been read in and interpreted, the consistency checking can begin.

The consistency checking portion of the program is composed of three major subroutines. First the activity and periodic function entities are checked to ensure that all the inputs and outputs referenced were indeed declared as input and output entities. Next the input and output entities are checked to see that each has been used by an activity or periodic function entity. In addition, the input and output entities have each of the structure types associated with them checked to be sure that the type has been defined. The third portion of the checking checks that each type has been used by an input or output, and then that each type is defined completely to a simple character string or set.

The report generated from the consistency checker program details the specific inconsistencies in the e-r-a requirements specification document. The three subroutines that perform the checking portion of the program generate the various inconsistency messages that comprise the report.

Testing of the consistency checking program was done using a

number of e-r-a requirement specification documents that conformed to the b-n-f grammar defined for the e-r-a document. Various test cases, that contained examples of each of the possible inconsistencies, were used to verify that the program handled each type of inconsistency defined. A sample specification containing no inconsistencies, was also used to test the program.

The source code for the consistency checker is in Appendix D. The program should be compiled using the standard C compiler resident on a UNIX operating system. The object module created from a C compile is the program to be executed with an e-r-a requirements specification document as input and output going directly to the terminal or redirected to a file.

For example; if a C compile of the source code created an executable object module called "checker" and an e-r-a requirements specification document (conforming to the b-n-f defined in Appendix A) was called "eraspec", the command:

```
checker < eraspec
```

would run the consistency checker against the "eraspec" input file and generate the inconsistency messages directly on the terminal. To direct the inconsistency messages to a file called, for example, "inconfile" (and using the same input file), the command would be:

```
checker < eraspec > inconfile
```

This could allow the user to keep a record of the inconsistencies.

The consistency checker program was designed to allow flexibility in the details used in defining the b-n-f for the e-r-a requirements specification document. The first section of the source code in Appendix D defines static constants for maximum variable name sizes, maximum internal table sizes and other static constant flags. The second section of the source code defines the actual entity keywords used and their corresponding attribute keywords used, as well as the maximum number of entities and attributes defined. In the third section of the source code the actual internal table structures are defined as well as global variables used within the program.

If the internal table structure is not large enough to hold an e-r-a requirements specification document, then the static constants defined in the first section of the code are the only items that need to be modified in the source code. The source code then only needs to be recompiled. If the entity or attribute keywords change then only the second section of the source code needs to be modified. Again, the source code would need to be recompiled.

If additional entities or attributes are defined, it may be necessary to modify the internal table structure defined in the third section of the source code. This could also require a modification in the routine(s) that collect the information that is stored in the internal tables.

Chapter Five

Conclusions and Extensions

The development of a software tool to check consistency of a requirements specification document can be a very useful program. The requirements specification phase, in the software development life cycle, needs to produce an accurate requirements specification document. An attribute of an accurate requirements specification document is consistency. By developing an automated method of ensuring a consistent document, the task of producing an accurate requirements specification document will become easier.

The consistency checker implementation provides a means to check consistency of the input and output entities. The inputs and outputs are cross checked with the activities and periodic functions. Those inputs and outputs that don't cross check generate an error message in the report indicating the applicable inconsistency. The inputs and outputs are then checked against the user defined type variables. Again, those that don't cross check generate an appropriate error message detailing the inconsistencies. Finally, the user defined type variables are checked to make sure that all that exist are used and completely defined.

The consistency checker works well in denoting the various inconsistencies it encounters in the interpretation of an entity-relationship-attribute (e-r-a) requirements specification document.

Checking for inconsistencies among the inputs and outputs is a very important part in obtaining a consistent requirements specification document. There are other aspects of the specifications document that could be checked for consistency.

The e-r-a specification format provides a number of relationships and attributes. Checking the consistency of the different relationships and attributes, that are applicable, is beyond the scope of this implementation. Checking the consistency of units, is one such possible attribute. The relationships subpart_is and subpart_of could be potentials for yet another implementation.

Extensions to this implementation could involve producing use count figures for the various inputs and outputs. While use counts don't directly relate to consistency, it would provide an indication of which inputs and outputs may be redundant and which may in essence be extraneous. The same information could be extracted for the user defined type variables.

Developing an automated method to identify inconsistencies among the inputs and outputs of a requirements specification document, is a means to enhance the total software development process.

References

[Ag82] Agusa, Kiyoshi, Atsushi Ohnishi and Yutaka Ohno, "Verification System for Formal Requirements Description", *IEEE Proceedings of 6th Conference on Software Engineering*, September 1982, Tokyo, Japan, pp. 120-126.

[Be76] Belford, P. C. and D. S. Taylor, "Specification Verification - A Key to Improving Software Reliability", *Proceedings of the Symposium on Computer Software Engineering*, Polytechnic Institute of New York, April 1976, pp. 83-96.

[Be77] Bell, Thomas E., David C. Bixler and Margaret 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.

[Bo75] Boehm, B. W., R. K. McClean and D. B. Urfrig, "Some Experience With Automated Aids to the Design of Large-Scale Reliable Software", *IEEE Proceedings International Conference on Reliable Software*, IEEE Catalog No. 75CH0940-7CSR, April 1975, pp. 105-113.

[Bo84] Boehm, Barry W., "Verifying and Validating Software Requirements and Design Specifications", *IEEE Software*, Vol. 1, No. 1, January 1984, pp. 75-88.

[Ca81] Campbell, R. H. and P. G. Richards, "SAGA: A system to Automate the Management of Software Production", *AFIPS Conference Proceedings, 1981 National Computer Conference*, 1981, pp. 231-234.

[Da79] Davis, Alan M. and Tomlinson G. Rauscher, "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications", *IEEE Proceedings Specification on Reliable Software*, 1979, pp. 15-35.

[Gr82] Greenspan, Sol J., John Mylopoulos and Alex Borgida, "Capturing More World Knowledge in the Requirements Specification", *IEEE Proceedings of 6th Conference on Software Engineering*, September 1982, Tokyo, Japan, pp. 225-234.

[He83] Heitmeyer, Constance L. and John D. McLean, "Abstract Requirements Specification: A New Approach and Its Application", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, September 1983, pp. 580-589.

[Ny83] Nyari, Erika and Harry Sneed, "SOFSPEC: A Pragmatic Approach to Automated Specification Verification", *Journal of Systems and Software*, Vol. 3, No. 3, September 1983, pp. 193-200.

[Pr82] Pressman, Roger S. *Software Engineering A Practitioner's Approach*, McGraw-Hill Book Company, New York, NY, 1982.

[Re79] Reifer, Donald J. and Stephen Trattner, "A Glossary of Software Tools and Techniques", *IEEE Tutorial: Automated Tools for Software Engineering, Compsac79*, Computer Society Press, 1979, pp. 6-14.

[Ro77] 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.

[Ru82] Rudmik, A., B. E. Casey and H. Cohen, "Consistency Checking Within Embedded Design Languages", *IEEE Proceedings of 6th Conference on Software Engineering*, September 1982, Tokyo, Japan, pp. 236-245.

[Te77] 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.

[Ti81] Tichy, Walter F., "Software Development Control Based on Module Interconnection", *IEEE Tutorial: Software Development Environments, Compsac81*, Computer Society Press, 1981, pp. 272-284.

[Tr82] Treleaven, Philip C. and Isabel Gouveia Lima, "Japan's Fifth-Generation Computer Systems", *IEEE Computer*, Vol. 15, No. 8, August 1982, pp. 79-88.

[Wa81] Wasserman, Anthony I., "Toward Integrated Software Development Environments", *IEEE Tutorial: Software Development Environments, Compsac81*, Computer Society Press, 1981, pp. 15-35.

[We82] Wertz, Harald, "The Design of an Integrated, Interactive and Incremental Programming Environment", *IEEE Proceedings of 6th Conference on Software Engineering*, September 1982, Tokyo, Japan, pp. 157-165.

[Ye83] Yeh, Raymond, "Software Engineering", *IEEE Spectrum*, Vol. 20, No. 11, November 1983, pp. 91-94.

Appendix A

B-N-F Grammar for E-R-A Requirements Specification Document

General Description

The era specification will consist of a set of frames. The order of the frame is not fixed. Each frame will contain information about one entity. Each frame will start on a newline. The first line in the frame will contain the keyword that describes the type of the entity and the name of the entity. The first letter in the type is capitalized. The type and the name are separated by a colon. At least one blank line will separate each frame.

The information in a frame is generally in the form of relations between this entity and other entities. Some of the information is in the form of attributes. An attribute gives information about this entity without referring to other entities. The order of these relations/attributes is not fixed.

Each relation/attribute is specified by a keyword that specifies the relation/attribute and its value. The value is either the name of the entity that has that relation or a text description of the attribute value. A colon separates the keyword and its value. Each relation/attribute starts on a new line. If a relation/attribute continues on to another line, the continuation line starts with a blank field followed by a colon. Multiple occurences of a relation/attribute is represented by multiple occurences of the keyword.

Entity Types

These entity types are not fixed. Additional entity types may be defined in the future. All entity types will start with a capital letter.

Activity
Type
Input
Output
Periodic_function
Input_output
Data
Constant
Comment
* additional entity types may be added at any time

Relations/Attributes

keywords
input
output
required_mode
necessary_condition
occurence
assertion
action
comment
media
structure
type
units
subpart_is
subpart_of
uses
* additional entity types may be added at any time

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> : <text> <text_cont>

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

<NL_B> ::=
        <NL> '    '
```

Lexical Scanner Information

Tokens used in the productions above may begin with <char> or one of the following characters: *$,':-={} Blanks can delimit tokens as well.

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 : Requirements specification for the chess program

Comment : as of 6/25/84 1:40 in ksu832:/usrb/we/era
        :
        : comment on specification:
        :    Not all of the activities necessary for this program
        : to be implemented are included in this description.
        : Some activities are not included if their activities
        : were determined by the other activities.  The activity
        : of interpreting the user's command was not included.

Type : $piece$
   structure : a string from the set {Kr,Kk,Kb,K,Q,Qb,Qk,Qr,p}

Type : $rank$
   structure : a string from the set {1,2,...8}

Type : $position$
   structure : $piece$ $rank$

Type : $piece_position$
   structure : $piece$ ',' $position$

Type : $board_matrix$
   structure : array[1..8,1..8] of $piece$ OR ' '

Input : $board_description$
   media     : keyboard
   structure : 'white' set of $piece_position$
   structure : 'black' set of $piece_position$
   structure : 'end'

Input : $name_of_game$
   media     : keyboard
   structure : 1 to 20 alphanumeric characters

Input : $new_user_input$
   media     : keyboard
   structure : any string

Input_output : $stored_board$
   media     : secondary_storage
   structure : information to recreate the board configuration

Input_output : $chess_board$
   media     : internal
```

```
   structure : $board_matrix$

Comment : This page contains those Input entities which are
        : directly related to a command which the user of
        : the chess game might enter.  (As opposed to Input
        : data which is not a command, ie: $name_of_game$).

Input : $move$
   media     : keyboard
   structure : 'm' $position$ '-' $position$

Input : $display_board$
   media     : keyboard
   structure : 'display'

Input : $create$
   media     : keyboard
   structure : 'create'

Input : $concede$
   media     : keyboard
   structure : 'concede'

Input : $store$
   media     : keyboard
   structure : 'store' $name_of_game$

Input : $retrieve$
   media     : keyboard
   structure : 'retrieve' $name_of_game$

Comment : The remaining Input entities are 'pseudo commands'
        : intended to aid in manually exercising
        : Periodic_functions.  The entities were named by
        : switching the first and last words so as not to
        : cause name collisions with the Output entities

Input : $mate_stale$
   media     : keyboard
   structure : 'stalemate'

Input : $limit_time$
   media     : keyboard
   structure : 'time_limit'

Input : $out_time$
   media     : keyboard
   structure : 'time_out'

Input : $check_input$
```

```
   media     : keyboard
   structure : 'input_check'

Comment : 1 Input entity above is unused.
        : 1 Input entity is omitted.

Output : $status$
   media     : crt
   structure : string from the set {'your move','check',
   structure : 'checkmate','concede'}

Output : $board_display$
   media     : crt
   structure : visually oriented display of current chess board

Output : $syntax_error$
   media     : crt
   structure : <cr> 'illegal, try again'

Output : $store_message$
   media     : crt
   structure : 'board stored'
   structure : 'storage failed'

Output : $retrieve_message$
   media     : crt
   structure : $name_of_game$ 'retrieved'
   structure : 'retrieval failed'

Output : $stalemate$
   media     : crt
   structure : 'stalemate occurred'

Output : $time_warning$
   media     : crt
   structure : 'this is a warning - 5 minutes elapsed'

Output : $time_out$
   media     : crt
   structure : 'too much time - game over'

Output : $move_message$
   media     : crt
   structure : <cr>
   structure : 'illegal move'

Output : $computer_move_message$
   media     : crt
   structure : 'computer moves from' $position$ 'to' $position$
```

```
Activity : Initialize_board
   keywords            : Standard_board,Initialize,Place_pieces
   input               : NONE
   output              : $chess_board$
   required_mode       : *START*
   necessary_condition : $start$
   assertion           : The output board is a correct representation of
                       : the standard starting configuration for chess

Activity : Create_special_board
   keywords            : Assign_positions,Place_pieces
   input               : $board_description$
   output              : $chess_board$
   required_mode       : *START*
   necessary_condition : $create$

Activity : Store_board
   keywords            : Store_game_status,Save_board
   input               : $name_of_game$
   input               : $chess_board$
   output              : $store_message$
   required_mode       : *NORMAL*
   necessary_condition : $store$
   assertion           : the game is stored in file '$name_of_game$'

Activity : Retrieve_board
   keywords            : Retrieve_board
   input               : $name_of_game$
   output              : $chess_board$
   output              : $retrieve_message$
   required_mode       : *START*
   necessary_condition : $retrieve$
   assertion           : Retrieves game stored in file '$name_of_game$'
                       : if successful

Comment : 1 Input item above is related to more than 1 other entity

Activity : Validate_user_move
   keywords            : Check_move,Check_m_status,Move_validation
   input               : $chess_board$
   input               : $move$
   output              : $move_message$
   required_mode       : *NORMAL*
   assertion           : If the move is illegal,
                       : the mode changes to *ILLEGAL*

Activity : Computer_Move
   comment             : used to be Move
   keywords            : Select_move,Select_status
   input               : $chess_board$
```

```
   output                : $chess_board$
   output                : $computer_move_message$
   output                : $status$
   required_mode         : *NORMAL*
   action                : mode may change to *END*
                         : if $status$ = 'checkmate' OR 'concede'

Activity : Update_board
   keywords              : Update_position,Update_status
   input                 : $chess_board$
   input                 : $move$
   output                : $chess_board$
   required_mode         : *NORMAL*

Activity : Display_board
   keywords              : Display
   input                 : $chess_board$
   output                : $board_display$
   required_mode         : *NORMAL*
   required_mode         : *END*
   necessary_condition   : $display_board$

Comment : 1 Input item above is related to more than 1 other entity

Comment : for simplicity, pseudo Input entities exist to
        : manually exercise these Periodic_functions.

Periodic_function : Stalemate
   required_mode : *NORMAL*
   occurence     : whenever a board congifuration is repeated 3 times
   input         : $mate_stale$
   output        : $stalemate$
   action        : change mode to *END*

Periodic_function : Time_Limit
   required_mode : *NORMAL*
   occurence     : whenever the user response time exceeds 5 minutes
   input         : $limit_time$
   output        : $time_warning$
   action        : NONE

Periodic_function : Time_Out
   required_mode : *NORMAL*
   occurence     : whenever the user response time exceeds 10 minutes
   input         : $out_time$
   output        : $time_out$
   action        : change mode to *END*

Periodic_function : Input_Check
   required_mode : every mode
```

```
   occurence      : whenever user input does not match allowed syntax
   input          : $check_input$
   output         : $syntax_error$
   action         : change mode to *ILLEGAL*


MODE_TABLE
   Mode : *ILLEGAL*
   Mode : *NORMAL*
   Mode : *START*
   Mode : *END*

   Initial_Mode : *START*

   Allowed_Mode_Transitions :
   $create$                                  : *START* -> *NORMAL*
   $start$                                   : *START* -> *NORMAL*
   $retrieve$                                : *START* -> *NORMAL*

   $status$ = 'checkmate'                    : *NORMAL* -> *END*
   $status$ = 'concede'                      : *NORMAL* -> *END*
   $stalemate$                               : *NORMAL* -> *END*
   $time_out$                                : *NORMAL* -> *END*

   $move_message$ = 'illegal move'           : *NORMAL* -> *ILLEGAL*
   $syntax_error$                            : *NORMAL* -> *ILLEGAL*

   $syntax_error$                            : *START* -> *ILLEGAL*

   $syntax_error$                            : *END* -> *ILLEGAL*

   $new_user_input$                          : *ILLEGAL* -> *NORMAL*

   $'<cr>'$                                  : *END* -> *START*

Comment : 2 of the above transitions are unfirable.
        : 3 of the above transitions cause mode
        : indeterminacy.
        : as specified here, *END* is not a terminal mode.
```

Appendix C

Module Specification of Consistency Checker

Static Constants:

Define standard C libraries to be included in source.
Define maximum size of variable names.
Define maximum size of internal tables.
Define constant "flag" values.

Keyword Values:

Define different entity keyword names.
Identify maximum number of entity keywords defined.
Define different attribute keyword names.
Identify maximum number of attribute keywords defined.

Internal Structures:

Declare structures for different internal tables used.
Declare global variables used in the program.
Declare and initialize counters used in the program.

main:

Functions:
Control syntax scanning of input document.
Call appropriate function based on entity encountered.
Print internal table values for debugging purposes.
Call checking functions.
Inputs:
e-r-a requirements specification input file
Outputs:
none
Global Variables:
ent_words, MAXENTS, line, MAXLINE

type_ent:

```
  Functions:
    Extract name and structure information for type entities.
    Store collected information in internal table "t_ents".
  Inputs:
    e-r-a input file
  Outputs:
    t_ents
  Global Variables:
    remain_line, INDXTYPE, MAXLINE
```

i_o_ent:

```
  Functions:
    Extract name and structure information for Input, Output
      and Data entities.
    Store collected information in internal table "i_o_ents".
  Inputs:
    e-r-a input file, Input/Output/Data flag
  Outputs:
    i_o_ents
  Global Variables:
    remain_line, INDXIO, MAXLINE
```

act_pf_ent:

```
  Functions:
    Extract input and output names from Activity and Periodic_
      function entities.
    Store collected information in internal table "a_f_ents".
  Inputs:
    e-r-a input file
  Outputs:
    a_f_ents
  Global Variables:
    INDXACTF
```

scanner:

Functions:
Perform lexical scanning to extract keywords and identify corresponding names and values to be placed in internal tables (t_ents, i_o_ents and a_f_ents).
Called by three entity functions (type_ent, i_o_ent and act_pf_ent).
Inputs:
e-r-a input file
Outputs:
(Returns) Attribute or Entity
Global Variables:
ent_att, last_att, att_words, line, MAXLINE, MAXATTS

print_tbls:

Functions:
Print internal type entity (t_ents) table.
Print internal input/output entity (i_o_ents) table.
Print internal activity/function entity (a_f_ents) table.
Inputs:
INDXTYPE, INDXIO, INDXACTF
Outputs:
Contents of Global Variables Tables
Global Variables:
t_ents, i_o_ents, a_f_ents, MAXSTRUCTS, MAXIOS

act_pf_check:

Functions:
Check that inputs defined for an activity or function are defined.
Check that outputs defined for an activity or function are defined.
Inputs:
none
Outputs:
Error Messages
Global Variables:
a_f_ents, i_o_ents, INDXACTF, INDXIO

i_o_check:

Functions:
Check that a defined input is used by an activity or function.
Check that a defined output is produced by an activity or function.
Check that all inputs and outputs are further defined.
Inputs:
none
Outputs:
Error Messages
Global Variables:
i_o_ents, t_ents, INDXIO, INDXTYPE

type_check:

Functions:
Check that all types are completely defined.
Check that all types are used to define an input, output or another type.
Inputs:
none
Outputs:
Error Messages
Global Variables:
t_ents, INDXTYPE

Appendix D

Source Code For Implementation

```
    /*  This program performs consistency checking between */
    /*  input and output entities in relation to activity  */
    /*  and periodic function entities of a requirements   */
    /*  specification document.                            */
    /*                                                     */
    /*  The input required for this progam in called an    */
    /*  ERA (entity-relationship-attribute) requirements   */
    /*  specification document.  This type of document is  */
    /*  a keyword format description of a software project.*/

#include <stdio.h>
#include <strings.h>


    /*  This section contains the static constants used     */
    /*  throughout the program.  Arbitrary numbers were     */
    /*  selected to indicate the number of entities         */
    /*  expected for each type of entity.  These numbers    */
    /*  are used to declare the sizes of the internal table */
    /*  structures.                                         */


#define MAXCHARS  40    /* maximum # of characters in entity or */
                        /*   attribute name                     */
#define MAXLINE  100    /* maximum # of characters in an input line */
#define MAXIOS  5       /* maximum # of inputs or outputs in an   */
                        /*   activity or periodic function entity */
#define MAXSTRUCTS  5   /* maximum # of structures for inputs, */
                        /*   outputs or types                  */
#define MAXTYPE  30     /* maximum # of type entities */
#define MAXACTF  30     /* maximum # of activities or periodic */
                        /*   functions                         */
#define MAXIO  50       /* maximum # of input, output,        */
                        /*   input_output and data entities   */
#define NL  '\n'
#define ENTITY  99      /* flag used to identify an entity */
#define TRUE  1
#define FALSE  0
#define DOLLAR '$'
```

```
    /*  This section contains the entity and attribute      */
    /*  names used specifically by the consistency checker. */
    /*  It is in a table format, so if changes are made     */
    /*  to the e-r-a specification format the corres-       */
    /*  ponding entries are all that need to be modified.   */


#define TYPE      '0'   /* Type Entity */
#define INPUT     '1'   /* Input Entity */
#define I_O       '2'   /* Input_output Entity */
#define OUTPUT    '3'   /* Output Entity */
#define DATA      '4'   /* Data Entity */
#define ACTIVITY  '5'   /* Activity Entity */
#define P_F       '6'   /* Periodic_function Entity */

#define MAXENTS   7     /* Maximum number of entities defined */


        struct entities {
                char *entword;
        }  ent_words[] = {
                "Type",
                "Input",
                "Input_output",
                "Output",
                "Data",
                "Activity",
                "Periodic_function"
           };

#define INPUTATT  '0'   /* input attribute */
#define OUTPUTATT '1'   /* output attribute */
#define STRUCTATT '2'   /* structure attribute */

#define MAXATTS   3     /* maximum number of attributes defined */


        struct atts {
                char *attword;
        }  att_words[] = {
                "input",
                "output",
                "structure"
           };
```

```
/*  This section contains the declarations for the    */
/*  internal data structures and the global variables */
/*  used throughout the program.                      */


    struct in_outs {
            char i_o_name[MAXCHARS];
    };

    struct act_func_tbl {
            char name[MAXCHARS];
            struct in_outs a_f_ins[MAXIOS];
            int num_ins;
            struct in_outs a_f_outs[MAXIOS];
            int num_outs;
    };

    struct fmt_struct {
            char struct_name[MAXCHARS];
    };

    struct i_o_tbl {
            char name[MAXCHARS];
            int iflag;
            int oflag;
            int itimes;
            int otimes;
            struct fmt_struct i_o_structs[MAXSTRUCTS];
    };

    struct type_tbl {
            char name[MAXCHARS];
            struct fmt_struct type_structs[MAXSTRUCTS];
            int used_cnt;
    };

    struct type_tbl t_ents[MAXTYPE];

    struct i_o_tbl i_o_ents[MAXIO];

    struct act_func_tbl a_f_ents[MAXACTF];

    char line[MAXLINE];
    char name[MAXLINE];
    char remain_line[MAXLINE];
    char ent_att[MAXCHARS];
    char colon[5];
```

```
char NONE[] = "NONE";
char null = '\0';
char colon_key[] = ":";

int n, s;
int last_att;
int cont_flag;
int INDXIO = -1;
int INDXTYPE = -1;
int INDXACTF = -1;
```

```
/*  This is the main program.  It controls the scanning */
/*  of the ERA input document and stores the necessary  */
/*  information, required for checking, into internal   */
/*  tables.  After the entire document has been read    */
/*  in, the various consistency checking routines are   */
/*  called to perform their checks.                     */


    main()
    {

    int i, MATCH;

      cont_flag = FALSE;
      s = scanf("%s%[ :]",ent_att,colon);

      while (s != EOF)
      {
        MATCH = FALSE;
        if (s == 2)
        {
          for (i=0;(i < MAXENTS) && !MATCH;i++)
            if (strcmp(ent_att,ent_words[i].entword) == 0)
                  MATCH = TRUE;
          if (MATCH)
          {
            s = scanf("%s",name);
            switch (--i + '0')
            {
              case TYPE:
                type_ent();
                break;
              case INPUT:
              case I_O:
              case OUTPUT:
              case DATA:
                i_o_ent(i + '0');
                break;
              case ACTIVITY:
              case P_F:
                act_pf_ent();
                break;
              default:
                break;
            }
            cont_flag = FALSE;
          }
```

```
        else
        {
          fgets(line,MAXLINE,stdin);  /* read rest of line */
          s = scanf("%s%[ :]",ent_att,colon);
        }
      }
      else
      {
        fgets(line,MAXLINE,stdin);  /* read rest of line */
        s = scanf("%s%[ :]",ent_att,colon);
      }
    }

    /* the next function will print the internal tables */
    print_tbls();

    act_pf_check();

    i_o_check();

    type_check();
}
```

```
/*  This function is called if the entity encountered  */
/*  is a "type" entity.  The name and structure        */
/*  information are retained for type entities.        */


    type_ent()
    {

    int i, k, dflag, ts;
    char *lptr;

      ts =0;
      strcpy(t_ents[++INDXTYPE].name,name);
      t_ents[INDXTYPE].used_cnt = 0;

      while ( s != EOF )
        {
            n = scanner();
            if (n == ENTITY) return;
            switch (n + '0')
            {
              case STRUCTATT:
                if ((lptr = fgets(remain_line,MAXLINE,stdin))
                    != NULL)
                {
                  dflag = FALSE;
                  k = 0;
                  t_ents[INDXTYPE].type_structs[ts].\
                    struct_name[0] = 'X';

                    /* initialize to somethine other than a */
                    /* $meta_variable$                      */

                  for (i=0;remain_line[i] != null;i++)
                  {
                    if ((remain_line[i] == DOLLAR) || dflag)
                    {
                      t_ents[INDXTYPE].type_structs[ts].\
                        struct_name[k++] = remain_line[i];
                      if ( !dflag ) dflag = TRUE;
                      else
                        if (remain_line[i] == DOLLAR)
                        {
                          t_ents[INDXTYPE].type_structs[ts].\
                            struct_name[k] = null;
                          dflag = FALSE;
                          ts++;
```

```
                        k = 0;
                    }
                }
            }
        }

        break;
        default:
        break;
    }
  }
}
```

```
/*  This function is called if the entity encountered  */
/*  is an "Input", "Output", "Input_output" or "Data"  */
/*  entity.  The name and structure information are    */
/*  retained for these kinds of entities.              */


    i_o_ent(c_flag)
    int c_flag;
    {

    char *lptr;
    int i, k, ios, dflag;

      ios = 0;
      strcpy(i_o_ents[++INDXIO].name,name);
      i_o_ents[INDXIO].itimes = 0;
      i_o_ents[INDXIO].otimes = 0;

      switch (c_flag)
      {

        case INPUT:
            i_o_ents[INDXIO].iflag = TRUE;
            i_o_ents[INDXIO].oflag = FALSE;
            break;

        case OUTPUT:
            i_o_ents[INDXIO].oflag = TRUE;
            i_o_ents[INDXIO].iflag = FALSE;
            break;

        case I_O:
        case DATA:
            i_o_ents[INDXIO].iflag = TRUE;
            i_o_ents[INDXIO].oflag = TRUE;
            break;

        default:
          break;
      }

      while ( s != EOF )
        {
            n = scanner();
            if (n == ENTITY) return;
            switch (n + '0')
            {
```

```
                case STRUCTATT:
                  if (( lptr = fgets(remain_line,MAXLINE,stdin))
                     != NULL)
                  {
                      dflag = FALSE;
                      k = 0;
                            i_o_ents[INDXIO].i_o_structs[ios].\
                              struct_name[0] = null;
                      for (i=0;remain_line[i] != null;i++)
                      {
                        if ((remain_line[i] == DOLLAR) || dflag)
                          {
                            i_o_ents[INDXIO].i_o_structs[ios].\
                              struct_name[k++] = remain_line[i];
                            if (!dflag) dflag = TRUE;
                              else
                              {
                                if (remain_line[i] == DOLLAR)
                                {
                                  i_o_ents[INDXIO].i_o_structs[ios].\
                                    struct_name[k] = null;
                                  dflag = FALSE;
                                  ios++;
                                  k = 0;
                                }
                              }
                          }
                      }
                  }

                  break;
                  default:
                  break;
              }
          }
      }
```

```
/*  This function is called if the entity encountered   */
/*  is an "Activity" or "Periodic_function" entity.     */
/*  The input and output names associated with these    */
/*  kinds of entities are retained.                     */


    act_pf_ent()
    {

    int afi, afo;

      afi = 0;
      afo= 0;
      strcpy(a_f_ents[++INDXACTF].name,name);
      a_f_ents[INDXACTF].num_ins = 0;
      a_f_ents[INDXACTF].num_outs = 0;

      while ( s != EOF )
      {
        n = scanner();
        if (n == ENTITY) return;
        switch (n + '0')
        {
            case INPUTATT:
              scanf("%s",name);
              if (strcmp(name,NONE) != 0)
              {
              strcpy(a_f_ents[INDXACTF].a_f_ins[afi++].i_o_name
                    ,name);
              a_f_ents[INDXACTF].num_ins++;
              }

            break;
            case OUTPUTATT:
              scanf("%s",name);
              if (strcmp(name,NONE) != 0)
              {
              strcpy(a_f_ents[INDXACTF].a_f_outs[afo++].i_o_name
                    ,name);
              a_f_ents[INDXACTF].num_outs++;
              }

            break;
            default:
            break;
        }
      }
```

```
  return;
}
```

```
/*  This function is used by the three functions --     */
/*  type_ent, i_o_ent and act_pf_ent to perform common */
/*  lexical scanning of the input file.                */


    scanner()
    {

    int i;

      while ((s=scanf("%s",ent_att)) != EOF)
      {
        if (ent_att[0] == NL) continue;

        else if((strcmp(ent_att,colon_key) == 0) && cont_flag)
            return(last_att);
        s = scanf("%[ :]",colon);
        if (s == 1)
        {
           s = 2;
               if (('A' <= ent_att[0]) && (ent_att[0] <= 'Z'))
                    return (ENTITY);
               else
               {
                    for (i=0;i < MAXATTS;i++)
                    if (strcmp(ent_att,att_words[i].attword)
                       == 0)
                    {
                        last_att = i;
                        cont_flag = TRUE;
                        return (i);
                    }
                    cont_flag = FALSE;
                    fgets(line,MAXILNE,stdin);
                      /* read rest of line */
               }
        }
        else fgets(line,MAXLINE,stdin);
          /* read rest of line */
      }
    }
```

```
/*  This function does the required checking for          */
/*  activity or periodic function entities.  The checks   */
/*  include items used that are not defined and items     */
/*  produced which are not defined.                       */


    act_pf_check()
    {

    int i, j, k, MATCH;

      for (i=0;i <= INDXACTF;i++)
      {
        MATCH = FALSE;
        for (j=0;j < a_f_ents[i].num_ins;j++)
        {
          for (k=0;(k <= INDXIO) && !MATCH;k++)
          {
            if (strcmp(a_f_ents[i].a_f_ins[j].i_o_name,
                i_o_ents[k].name) == 0)
            {
              MATCH = TRUE;
              i_o_ents[k].itimes++;
            }
          }
          if (!MATCH)
            {
            printf("ACTIVITY %s USES %s WHICH IS NOT DEFINED",
              a_f_ents[i].name,a_f_ents[i].a_f_ins[j].i_o_name);
            printf(" AS AN input OR data item \n");
            }
          MATCH = FALSE;
        }

        MATCH = FALSE;
        for (j=0;j < a_f_ents[i].num_outs;j++)
        {
          for (k=0;(k <= INDXIO) && !MATCH;k++)
          {
            if (strcmp(a_f_ents[i].a_f_outs[j].i_o_name,
                i_o_ents[k].name) == 0)
            {
              MATCH = TRUE;
              i_o_ents[k].otimes++;
            }
          }
          if (!MATCH)
```

```
            {
            printf("ACTIVITY %s PRODUCES %s WHICH IS NOT",
            a_f_ents[i].name,a_f_ents[i].a_f_outs[j].i_o_name);
            printf(" DEFINED AS AN output OR data item \n");
            }
          MATCH = FALSE;
        }

    }
}
```

```
/*  This function does the required checking for input */
/*  and output entities.  There are a number of checks */
/*  performed in this function.  A check is made to    */
/*  ensure an item is an input to an activity or       */
/*  function.  A check is made to ensure an item is an */
/*  output from an activity or function.  A check is   */
/*  made to decompose types to see that they are       */
/*  defined by another input or output entity.         */


i_o_check()
{

int i, j, k, FOUND;

  for (i=0;i <= INDXIO;i++)
  {
    if (i_o_ents[i].iflag && i_o_ents[i].oflag)
      if ((i_o_ents[i].itimes == 0) || (i_o_ents[i].otimes == 0))
        printf("%s NOT AN input OR output TO/FROM ANY ACTIVITY \n",
          i_o_ents[i].name);
      else;
    else if (i_o_ents[i].iflag && (i_o_ents[i].itimes == 0))
           printf("%s NOT USED AS AN input TO ANY ACTIVITY \n",
             i_o_ents[i].name);
         else if (i_o_ents[i].oflag && (i_o_ents[i].otimes == 0))
           printf("%s NOT AN output FROM ANY ACTIVITY \n",
             i_o_ents[i].name);
    for (k=0;i_o_ents[i].i_o_structs[k].struct_name[0] != null;k++)
    {
      FOUND = FALSE;
      if (i_o_ents[i].i_o_structs[k].struct_name[0] == DOLLAR)
      {
        for (j=0;(j <= INDXTYPE) && !FOUND;j++)
        {
          if (strcmp(t_ents[j].name,
              i_o_ents[i].i_o_structs[k].struct_name) != 0)
            continue;
          else
          {
            t_ents[j].used_cnt++;
            FOUND = TRUE;
          }
        }
        if (!FOUND)
        {
           printf("%s NOT DEFINED AS A type \n",
```

```
                i_o_ents[i].i_o_structs[k].struct_name);
                for (j=0;(j <= INDXIO) && !FOUND;j++)
                {
                  if (strcmp(i_o_ents[j].name,
                    i_o_ents[i].i_o_structs[k].struct_name) != 0)
                      continue;
                  else FOUND = TRUE;
                }
                if (!FOUND)
                   printf("   AND %s IS NOT DEFINED AS ANOTHER\
                              INPUT OR OUTPUT \n",
                   i_o_ents[i].i_o_structs[k].struct_name);
                else printf("   BUT %s IS DEFINED BY ANOTHER\
                              INPUT/OUTPUT ENTITY \n",
                   i_o_ents[i].i_o_structs[k].struct_name);
            }
        }
    }
  }
}
```

```
/*  This function does the required checks for type   */
/*  entities.  A check is made to see if a type is    */
/*  defined to it's lowest element.  All types are    */
/*  checked to make sure they are used by either an   */
/*  input or output entity.                           */


type_check()
{

int i, j, k, FOUND;

  for (i=0;i <= INDXTYPE;i++)
    for (j=0;t_ents[i].type_structs[j].struct_name[0] != null;j++)
      if (t_ents[i].type_structs[j].struct_name[0] == DOLLAR)
      {
        FOUND = FALSE;
        for (k=0;(k <= INDXTYPE) && !FOUND;k++)
        {
          if (strcmp(t_ents[i].type_structs[j].struct_name,
                t_ents[k].name) != 0) continue;
          else
          {
            t_ents[k].used_cnt++;
            FOUND = TRUE;
          }
        }
        if (!FOUND)
           printf("%s type NOT DEFINED BY ANOTHER type \n",
             t_ents[i].type_structs[j].struct_name);
      }

  for (i=0;i <= INDXTYPE;i++)
    if (t_ents[i].used_cnt == 0)
        printf("%s type NOT USED AS AN input OR output \n",
          t_ents[i].name);
}
```

```
/*  This function is merely used to display the contents */
/*  of all the internal tables used to store the         */
/*  scanned and collected information from the input.    */


    print_tbls()
    {

    int i, j;

      for (i=0;i <= INDXTYPE;i++)
      {
        printf("type name %s \n",t_ents[i].name);
          for (j=0;j < MAXSTRUCTS;j++)
            printf("  type struct name %s \n",
              t_ents[i].type_structs[j].struct_name);
      }

      for (i=0;i <= INDXIO;i++)
      {
        printf("i/o name %s \n",i_o_ents[i].name);
          for (j=0;j < MAXSTRUCTS;j++)
            printf("  i/o struct name %s \n",
              i_o_ents[i].i_o_structs[j].struct_name);
      }

      for (i=0;i <= INDXACTF;i++)
      {
        printf("a/f name %s \n",a_f_ents[i].name);
          for (j=0;j < MAXIOS;j++)
          {
            printf("  a/f ins i/o name %s \n",
              a_f_ents[i].a_f_ins[j].i_o_name);
            printf("  a/f outs i/o name %s \n",
              a_f_ents[i].a_f_outs[j].i_o_name);
          }
      }
    }
```

THE IMPLEMENTATION OF AN INPUT/OUTPUT CONSISTENCY CHECKER FOR A REQUIREMENTS SPECIFICATION DOCUMENT

by

LAURA HAZEL WELMERS

B.S., Michigan State University, 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

1985

ABSTRACT

A requirements specification document lays the foundation in the creation of a computer-based software development project. Checking for consistency within a requirements specification document is a means to ensuring its validity. This paper describes the implementation of a program to check the consistency of an entity-relationship-attribute (e-r-a) requirements specification document. The consistency checker checks input and output entities in relation to activity and function entities. It ensures that all input and output entities are used in an activity or function as well as that all inputs and outputs of activity or function entities are correspondingly defined as such entities. The structure attributes of input and output entities are also checked for consistency. A report is generated for each requirement specification document checked, detailing the inconsistencies encountered.