CONSISTENCY CHECKING OF REQUIREMENTS SPECIFICATIONS USING STRUCTURED ANALYSIS DIAGRAMS

by

AARON NOBLE FRIESEN

B.S., Kansas State University, 1985

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:

[illegible]

Dr. David A. Gustafson

8
4
.5C
37
74
2

TABLE OF CONTENTS

LIST OF FIGURES

Chapter One

REQUIREMENTS SPECIFICATIONS FOR SYSTEM DEVELOPMENT

Requirements specifications are the basis for developing a system. The requirements specification should define the problem and outline the characteristics (including constraints) of a correct solution, encompassing "everything necessary to lay the groundwork for subsequent stages in system development" [Ro77c]. To achieve this goal, these specifications must answer all questions that arise about what the system should do when completed. If a problem is well-defined in the requirements specification, the task of developing a solution becomes much easier.

Many specification methodologies exist for use in requirements specification. Some of the current methodologies are E-R-L, PSL/PSA, SADT, and TAGS. E-R-L (Entity-Relationship-Level) [Gu84] is a model based on an entity-relationship viewpoint. The E-R-L model uses frames for entities and relationships between entities, and includes the ability to have abstraction levels and meta-information. The implementation of various automated support tools has been planned, with a frame-editor currently in operation.

PSL/PSA (Problem Statement Language/Problem Statement Analyzer) [Te77] is a computer-aided structured documentation and analysis system. It uses a fixed set of objects and strongly typed relationships between objects. Throughout the specification process, textual information is entered into a database, where it can be accessed for analysis to produce various reports dealing with such things as object usage, system hierarchy, and modification diagnostics.

SADT (Structured Analysis and Design Technique) [Ro85] is a graphic, hierarchical dataflow model. SADT combines graphic language primitives with natural language to produce a hierarchical model with abstraction levels. At present, attempts are in progress to develop graphic automated support tools.

Finally, TAGS (Technology for the Automated Generation of Systems) [Si85] is a system that combines an Input/Output Requirements Language, with a system/software computer-based tool system. TAGS combines dataflow information along with control and timing information within a hierarchy of diagrams. This information, when accessed through the system database, enables error checking and system simulation.

Today, dataflow models are among the most popular in use for requirements specification. Dataflow models are popular because they are "very well suited for modeling the structure and behavior of most human organizations" [Rm85].

Structured Analysis (which is part of SADT) is a well-known example of a dataflow model.

STRUCTURED ANALYSIS DIAGRAMS

Structured Analysis diagrams are a requirements specification tool for developing large scale systems. Structured Analysis diagrams combine the conciseness of a graphic system with the expressiveness of a natural or formal language embedded within the diagrams [Ro85]. The choice of embedded language is specific to the type of system being developed. By having the ability to incorporate any embedded language, Structured Analysis diagrams are a specification tool that is "universal and unrestricted," making Structured Analysis diagrams a <u>domain-independent</u> system model [Ro77b]. Structured Analysis diagrams are a means of precisely specifying a system, analogous to industrial blueprints [Rm85]. The concise and complete combination of word and picture documentation enables the "rigorous expression of high-level ideas that previously had seemed too nebulous to treat technically" [Ro85]. The requirements specification begins at a high level of abstraction. Through decomposition, the system is broken down into a hierarchically related set of diagrams.

System complexity is managed in Structured Analysis by restricting a diagram to six or fewer parts. The notation used in Structured Analysis decomposition is very straight forward. Each of the six or fewer parts is represented as a single box. The left side of the box shows all inputs to the box, the right side shows all outputs from the box, the top shows controls, and the bottom shows mechanisms. The outputs are transformed from the inputs under the direction of the control, and the mechanism is the means of the transformation. The inputs, outputs, controls, and mechanisms are represented by arrows, which connect the various boxes, thus indicating relationships between the boxes [Ro77b]. When combined, the boxes and arrows form a <u>detail diagram</u>. The top-level detail diagram must completely encompass the breadth of the system.

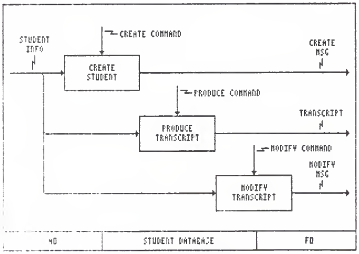


FIGURE 1-1 — STUDENT DATABASE

Figure 1-1 shows a possible decomposition for a simple student database for use in managing student transcripts. The system has three major acitivites: CREATE STUDENT, PRODUCE TRANSCRIPT, and MODIFY TRANSCRIPT. One input, STUDENT INFORMATION, is required for the system, with three system commands, CREATE, PRODUCE, and MODIFY, controlling the transformation of the input into the various outputs, CREATE MESSAGE, TRANSCRIPT, and MODIFY MESSAGE.

HIERARCHY IN STRUCTURED ANALYSIS DIAGRAMS

If any of the parts contained within the detail diagram are not fully specified, the decomposition process continues. The decomposition process forms a hierarchy of diagrams. Each box that is further decomposed is known as a _parent box_, and the diagram in which it is originally located is known as the _parent diagram_. The parts of a parent box are placed in a separate detail diagram, once again with six or fewer boxes. This new detail diagram is an in-depth description of the parent box from which it is derived, and encompasses the breadth of the parent box. For any part that still requires further specification, the hierarchical decomposition continues [Ro77b]. When the decomposition is complete, the set of diagrams will encompass the depth of the system, with each complete abstraction level in the hierarchy encompassing the breadth of the system.

Figures 1-2, 1-3, and 1-4 show a possible hierarchical decomposition of the parent diagram in figure 1-1. The CREATE STUDENT activity is detailed in figure 1-2, the PRODUCE TRANSCRIPT activity is detailed in figure 1-3, and the MODIFY TRANSCRIPT activity is detailed in figure 1-4.

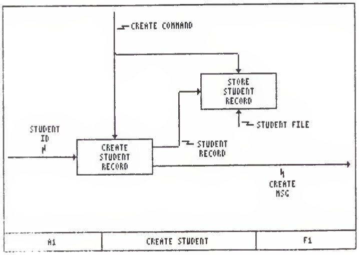


FIGURE 1-2 — CREATE STUDENT

The abstraction process that gives Structured Analysis much of its power can also cause a problem: inconsistency in naming information at different abstraction levels. Except for the most trivial of systems, a Structured Analysis specification will contain numerous diagrams. This introduces the possibility of naming inconsistencies across diagram boundaries.

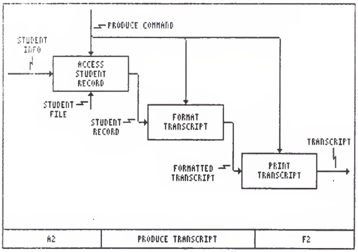


FIGURE 1-3 — PRODUCE TRANSCRIPT

CONSISTENCY IN SPECIFICATIONS

The requirements specification's main task is "to be able to answer questions" [Gu84], but an inconsistent specification is unable to perform this task because the specification contains contradictions. When examined as a whole, the various parts of a consistent requirements specification will not contradict one another [Rm85].

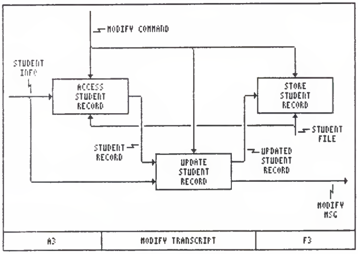


FIGURE 1-4 — MODIFY TRANSCRIPT

When working with a hierarchical methodology such as Structured Analysis, one area where inconsistencies are prevalent is where information crosses between levels of the system. In Structured Analysis diagrams, inputs, outputs, and controls are in this category. Specifically, the inputs (and also outputs and controls) of a detail diagram must match those from the parent box at the next higher level in the model.

As an example of this problem, examine figures 1-1 and 1-2. In figure 1-1, activity CREATE STUDENT requires one input: STUDENT INFORMATION. In figure 1-2, this input has been changed to read STUDENT ID. When examined individually, the diagrams seem to be correct; but when examined together, it can be shown that an inconsistency has already been introduced at the first level in the decomposition hierarchy. This problem increases as the size of the specified system increases and can become worse when different people specify different parts of the system.

The requirements specification should be analyzable for consistency. In fact, consistency checking "presuppose(s) the analyzability of the requirements by (various) means," either manually, or by automated tools [Rm85]. To make analysis possible, the requirements specification must be formalized. Furthermore, with more formality, it becomes more likely that the analysis can and will be performed by mechanical means [Rm85]. Mechanical analysis is advantageous since automated tools can enable easier and more accurate analysis. However, the right kind of information must be embedded within the formalized specification to enable computer tools to ensure consistency [Ro77c]. This information will actually be meta-information (information about information) and is usually included in the specification through the introduction of formal notations or possibly even a meta-language to aid in consistency checking.

Chapter Two

A NAMING CONVENTION TO AID IN THE CONSISTENCY CHECKING OF STRUCTURED ANALYSIS REQUIREMENTS SPECIFICATIONS

A consistent requirements specification is a necessity when developing a system. The requirements specification lays the groundwork for all subsequent stages. Without a strong foundation, it is unlikely that a correct solution can be completed for a problem; and if a correct solution is implemented, it is likely that the cost of development will be higher than necessary. Therefore, by reducing the number of errors in a system early in the development process, the probability of a correct solution, and a solution with less cost, is increased. Consistency checking of requirements specifications is one method of possibly reducing the number of errors in the implementation of a system.

INCONSISTENCIES WITHIN STRUCTURED ANALYSIS DIAGRAMS

In a Structured Analysis dataflow diagram, inconsistencies arise within the data elements that cross diagram boundaries. The number of diagrams in the specification of a complex system is large. The diagrams are usually developed manually. Many different people each develop small pieces of the system. Combining the large number of diagrams with current methods of development provides ample opportunities for inconsistencies to be introduced through miscommunication between developers, or simply through slight carelessness in recording the specification. As the system specification is decomposed, the information contained within a single diagram becomes more concrete. Abstract names given to data elements at a higher level in the specification will no longer be appropriate for the data elements at a lower, less abstract level. The names of data elements change to allow more information to be communicated. However, the changes introduced must be consistent with the information given in the next higher abstraction level.

To enable consistency checking, the specification must be formalized in some manner. This is usually done by embedding meta-information, or by adding notation within the existing system. The meta-information, or added notation, enables consistency checking by supplying needed information for stating intended relationships between the various parts of the specification.

The consistency checker, whether man or machine, then extracts the information and analyzes it by comparing the information from the specification with the expected results. In Structured Analysis, the extracted information must deal with how data elements are related between abstraction levels. Any differences between the extracted information and expected results indicates possible problems that may require correction or modification.

The consistency checks to be performed, will determine whether all data elements have their appropriate sources and sinks. This means that not only must a data element have a source and sink, but that same data element must logically have the same source and the same sink at all levels of abstraction. As the data element is decomposed, the relationship between abstraction levels in the decomposition must be shown. Therefore, an inconsistency is one of two things: 1) different data element names at adjacent abstraction levels for an identical data element, or 2) different sources or sinks at adjacent abstraction levels for an identical data element.

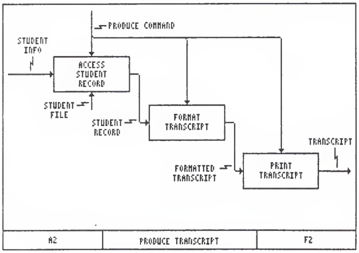


FIGURE 2-1 — ORIGINAL DIAGRAM FORM

A NOTATION ENABLING CONSISTENCY CHECKING

One possible added notation for locating the above inconsistencies is illustrated in Figures 2-1 and 2-2. The notation enables a consistency checker to follow the derivation of a data element. An extension is added to the element name, indicating its source (where it is obtained) and also its sink (where it is used). Figure 2-1 illustrates a complete diagram from a system specification in its original form. All input, output, and control identifiers are included in the illustration. Figure 2-2 shows the same basic diagram, with names removed and identifier extensions added. The extensions would actually be appended to the end of each data element, but appear in the diagram separately for clarity.

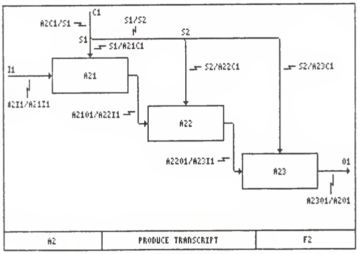


FIGURE 2-2 — DATA ELEMENT EXTENSIONS

The extensions specify the data element source and also the data element sink. For example, the data element STUDENT INFO in PRODUCE TRANSCRIPT begins as input one (I1) of this diagram (A2) and ends as input one (I1) of activity ACCESS STUDENT RECORD (A21). The extension therefore becomes A2I1/A21I1. The data element source identifier (the extension before the slash) and the data element sink identifier (the extension after the slash) are each made by either concatenating the activity identifier (i.e., 'A2') with the data element identifier (i.e., 'I1'), or for data elements being split or joined, from the arbitrarily assigned split/join identifier (i.e., 'S3'). Activity identifiers are taken from the identifier of the source/sink activity. For sources/sinks that cross diagram boundaries, the activity identifier is taken from the diagram identifier. Data element identifiers are assigned arbitrarily at the time of specification development. The relative numbering of data element identifiers must remain identical between abstraction levels, to reduce errors identified during consistency checking. Split/Join identifiers are assigned arbitrarily at the time extensions are added to data element identifiers. For an exact explanation of the method for adding extensions to data elements, see appendix B.

Current Structured Analysis specification styles allow for the logical splitting or joining of data elements at the diagram boundary, without explicitly specifying the split or join. Because of the method of source/sink identification, further formalization is required within a system specification. In addition to the extension notation, it becomes necessary to require that all data elements crossing diagram boundaries be in the same form on both sides of the boundary. Requiring that all splits and joins be explicity specified enables consistency checking.

THE METHOD OF CHECKING FOR CONSISTENCY

With the introduction of the above notation, it becomes possible to check for consistency within a requirements specification. Currently, structured analysis diagrams are produced manually, with limited aid from automated graphics systems. This means that the addition of the notation and the extraction of the information required for checking must also be done manually. The information must be gathered and arranged manually. To enable automated consistency checking, the information must be combined into a textual diagram description. The textual diagram description contains all necessary information required by the consistency checker.

The diagram description can be broken down into four basic parts: the IOC section, the activity section, the split section, and the join section. The IOC section specifies all inputs, outputs, and controls that cross the diagram boundary of a detail diagram to the abstraction level directly above the diagram. The activity section specifies all inputs, outputs, and controls that cross the diagram boundary to the abstraction level directly below the diagram. The split section and join section supply information for connecting data elements between the other two sections. Figure 2-3 at the end of the chapter shows a completed diagram description for the diagrams in figures 2-1 and 2-2.

DATA SOURCES AND SINKS

When diagram descriptions have been completed, the consistency checking can begin. Two types of checking must be performed: inter-diagram checks, and also intra-diagram checks. The inter-diagram checks ensure consistency between various levels within the diagram hierarchy. The intra-diagram checks ensure consistency within a single diagram.

Inter-diagram consistency checking must be performed for each activity in the diagram hierarchy that is decomposed in a lower-level detail diagram. Data sources are extracted from each activity and its related detail diagram. The possible data sources are inputs and controls to the activity, and the outputs from the detail diagram. For each source, the extension is used to locate the appropriate data sink. The possible data sinks are inputs and controls to the detail diagram, and outputs from the activity. The extension gives the diagram identifier and data element identifier. If a data element is found in the appropriate position, the data elements are compared. Non-identical data element names signify inconsistencies.

After all possible sources have been identified and checked, data element sinks are located. All sinks not matched to a data element source are additional inconsistencies.

Intra-diagram consistency checking is performed for each diagram in the diagram hierarchy. Intra-diagram checks are easier to perform than inter-diagram checks since all required information is located on one diagram. The process begins by extracting data sources. The possible data sources are inputs and controls to the diagram, and the outputs from each activity in the diagram. For each source, the extension is used to locate the appropriate data sink. The possible data sinks are inputs and controls to each activity in the diagram, and outputs from the diagram. Splits/Joins are treated similar to activities. If a data element is found in the appropriate position, the data elements are compared. Non-identical data element names signify inconsistencies. After all possible sources have been identified and checked, data element sinks must be located. All sinks not previously matched to a data element source are additional inconsistencies.

```
diagram:  PRODUCE_TRANSCRIPT

  input:  STUDENT_INFO.A2I1/A21I1
  control:  PRODUCE_COMMAND.A2C1/J1
  output:  TRANSCRIPT.A23O1/A2O1

  activity:  ACCESS_STUDENT_RECORD

    input:  STUDENT_INFO.A2I1/A21I1
    control:  PRODUCE_COMMAND.J1/A21C1
    output:  STUDENT_RECORD.A21O1/A22I1

  activity:  FORMAT_TRANSCRIPT

    input:  STUDENT_RECORD.A21O1/A22I1
    control:  PRODUCE_COMMAND.J2/A22C1
    output:  FORMATTED_TRANSCRIPT.A22O1/A23I1

  activity:  PRINT_TRANSCRIPT

    input:  FORMATTED_TRANSCRIPT.A22O1/A23I1
    control:  PRODUCE COMMAND.J2/A23C1
    output:  TRANSCRIPT.A23O1/A2O1

  split:  PRODUCE_COMMAND.A2C1/J1

    output:  PRODUCE_COMMAND.J1/A21C1,
             PRODUCE_COMMAND.J1/J2

  split:  PRODUCE_COMMAND.J1/J2

    output:  PRODUCE_COMMAND.J2/A22C1,
             PRODUCE_COMMAND.J2/A23C1

end:
```

FIGURE 2-3 DIAGRAM DESCRIPTION EXAMPLE

Chapter Three

CONSISTENCY CHECKER REQUIREMENTS SPECIFICATION

When specifying a system, several basic questions should be addressed: 1) What is the general description of the problem to be solved, 2) Who will be the predominant users of the system, 3) What is the required form for the system input, 4) What is the required form for the system output, and 5) What operational constraints exist for the system. After these questions have been answered, a system model must be developed.

GENERAL PROBLEM DESCRIPTION

The system to be implemented will take a requirements specification in the form of a set of structured analysis diagram descriptions and produce a report of any inconsistencies in inputs, outputs, and controls that occur within the specification. The consistency checker will check for any data source that does not have appropriate data sinks. When analyzing a single diagram, the possible sources are diagram inputs, diagram controls, and activity outputs; and the possible sinks are diagram outputs, activity inputs, and activity controls. When analyzing the diagram tree, the possible sources are activity inputs (taken from the parent dia- gram), activity controls (taken from the parent diagram), and diagram outputs, and the possible sinks are activity outputs (taken from the parent diagram), diagram inputs, and detail controls.

INVOCATION OF TOOL

The predominant users of the consistency checking system will be students in the Department of Computer Science at Kansas State University. It will be used along with other software tools at the university, and it would therefore be advantageous to operate similar to other available software tools. With this in mind, the invocation process should be similar to other applications on the targeted hardware. The invocation includes the mechanisms for obtaining input and directing output from the consistency checker.

DEFINITION OF INPUT

The required system input will be a set of textual structured analysis diagram descriptions. Each diagram description includes all information required by the consistency checker. This information states the name of the diagram; a list of all diagram inputs, outputs, and controls; a list of all activities within the diagram, along with the inputs, outputs, and controls to each activity; and also a list of all splits and joins of each data element within the diagram. The exact form for the diagram descriptions can be found in Appendix A. An example of a complete diagram description can be found later in this chapter, and instructions for description development can be found in Appendix B. The diagram description is in free-format, thus relieving the user from the problem of errors introduced by requiring highly structured, error-prone, formatted input.

DEFINITION OF OUTPUT

The output will include an echo of the input, producing a formatted copy, indenting sub-sections, and aligning columns of information. For any error encountered while scanning the input file, an appropriate error message will be issued, indicating the error encountered, and also its location within the input file. Also included in the output will be a report of all consistency errors found within the set of diagram descriptions. The list of all possible error messages are listed appendix D.

OPERATIONAL CONSTRAINTS

Operationally, the consistency checking system should perform in a manner similar to other available tools. The method for acquiring system input and producing system output will therefore be logical and familiar to the user.

STRUCTURED ANALYSIS MODEL OF THE PROPOSED SYSTEM

Figure 3-1 shows the system described previously in the General Problem Description and is entitled CONSISTENCY

CHECKER. The system contains three major activities: SCAN DIAGRAM DESCRIPTION INPUT, PRODUCE DIAGRAM DESCRIPTION OUTPUT, and PRODUCE CONSISTENCY REPORT. SCAN DIAGRAM DESCRIPTION INPUT scans the diagram description input producing either error messages relating to the input scan, or producing an internal representation of the diagram description that will be formatted and sent to output by PRODUCE DIAGRAM DESCRIPTION OUTPUT. The internal representation of the diagram description is also used by PRODUCE CONSISTENCY REPORT. This activity performs the consistency check producing appropriate error messages relating to the consistency state of the diagram descriptions. Further decomposition of the system leads to the specifications in figures 3-2 through 3-5.

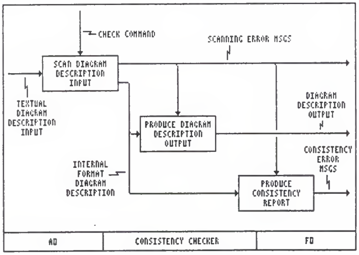

CHECK COMMAND
SCANNING ERROR MSGS
SCAN DIAGRAM DESCRIPTION INPUT
TEXTUAL DIAGRAM DESCRIPTION INPUT
DIAGRAM DESCRIPTION OUTPUT
PRODUCE DIAGRAM DESCRIPTION OUTPUT
CONSISTENCY ERROR MSGS
INTERNAL FORMAT DIAGRAM DESCRIPTION
PRODUCE CONSISTENCY REPORT
A0
CONSISTENCY CHECKER
F0

FIGURE 3-1 — CONSISTENCY CHECKER

Figure 3-2 describes the activity SCAN DIAGRAM DESCRIPTION INPUT, from diagram CONSISTENCY CHECKER. In this diagram, SCAN DIAGRAM DESCRIPTION INPUT is decomposed into GET TOKEN, CHECK SYNTAX, and STORE TOKEN. GET TOKEN obtains a single token from the input, making it available to CHECK SYNTAX. For tokens that are not within the required syntax, the token location is noted and an appropriate error message is issued. Tokens adhering to the required syntax are stored by STORE TOKEN in an internal representation of the diagram description for later access during consistency checking.

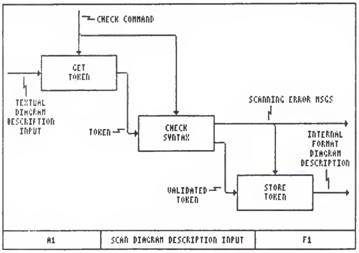

CHECK COMMAND
GET TOKEN
TEXTUAL DIAGRAM DESCRIPTION INPUT
TOKEN
CHECK SYNTAX
SCANNING ERROR MSGS
INTERNAL FORMAT DIAGRAM DESCRIPTION
VALIDATED TOKEN
STORE TOKEN
A1
SCAN DIAGRAM DESCRIPTION INPUT
F1

FIGURE 3-2 — SCAN DIAGRAM DESCRIPTION INPUT

Figure 3-3 describes the activity PRODUCE CONSISTENCY REPORT, from diagram CONSISTENCY CHECKER. In this diagram, PRODUCE CONSISTENCY REPORT is decomposed into LINK DESCRIPTION HIERARCHY, CHECK INTER-DIAGRAM CONSISTENCY, and CHECK INTRA-DIAGRAM CONSISTENCY. LINK DESCRIPTION HIERARCHY connects the descriptions from the free-format input into a tree of descriptions, checking that a single description is at the top of the structure, and allowing for further consistency checks. CHECK INTER-DIAGRAM CONSISTENCY checks for existing inconsistencies between related descriptions, with CHECK INTRA-DIAGRAM CONSISTENCY checking for existing inconsistencies within an individual description.

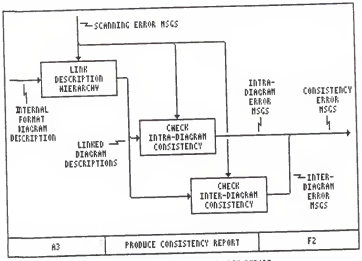
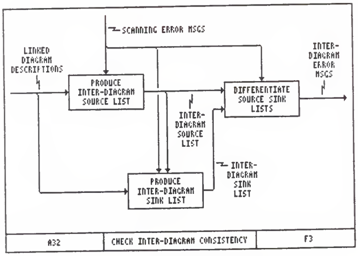

SCANNING ERROR MSGS
LINK DESCRIPTION HIERARCHY
INTERNAL FORMAT DIAGRAM DESCRIPTION
LINKED DIAGRAM DESCRIPTIONS
CHECK INTRA-DIAGRAM CONSISTENCY
INTRA-DIAGRAM ERROR MSGS
CONSISTENCY ERROR MSGS
CHECK INTER-DIAGRAM CONSISTENCY
INTER-DIAGRAM ERROR MSGS
A3
PRODUCE CONSISTENCY REPORT
F2

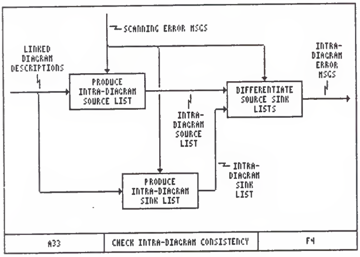
FIGURE 3-3 — PRODUCE CONSISTENCY REPORT

Figure 3-4 describes the activity CHECK INTER-DIAGRAM CONSISTENCY, from diagram PRODUCE CONSISTENCY REPORT. In this diagram, CHECK INTER-DIAGRAM CONSISTENCY is decomposed into PRODUCE INTER-DIAGRAM SOURCE LIST, PRODUCE INTER-DIAGRAM SINK LIST, and DIFFERENTIATE SOURCE/SINK LISTS. PRODUCE INTER-DIAGRAM SOURCE LIST accumulates all entities crossing a diagram boundary that are sources of data. PRODUCE INTER-DIAGRAM SINK LIST similarly accumulates all entities crossing a diagram boundary that are sinks of data. DIFFERENTIATE SOURCE/SINK LISTS compares the information in both lists, matching sources with all appropriate sinks, identifying all unmatched sources or sinks, along with their locations within the set of diagram descriptions.



SCANNING ERROR MSGS
LINKED DIAGRAM DESCRIPTIONS
PRODUCE INTER-DIAGRAM SOURCE LIST
DIFFERENTIATE SOURCE SINK LISTS
INTER-DIAGRAM ERROR MSGS
INTER-DIAGRAM SOURCE LIST
INTER-DIAGRAM SINK LIST
PRODUCE INTER-DIAGRAM SINK LIST
A32
CHECK INTER-DIAGRAM CONSISTENCY
F3

FIGURE 3-4 — CHECK INTER-DIAGRAM CONSISTENCY



SCANNING ERROR MSGS
LINKED DIAGRAM DESCRIPTIONS
PRODUCE INTRA-DIAGRAM SOURCE LIST
DIFFERENTIATE SOURCE SINK LISTS
INTRA-DIAGRAM ERROR MSGS
INTRA-DIAGRAM SOURCE LIST
INTRA-DIAGRAM SINK LIST
PRODUCE INTRA-DIAGRAM SINK LIST
A33
CHECK INTRA-DIAGRAM CONSISTENCY
F4

FIGURE 3-5 — CHECK INTRA-DIAGRAM CONSISTENCY

Figure 3-5 describes the activity CHECK INTRA-DIAGRAM CONSISTENCY, from diagram PRODUCE CONSISTENCY REPORT. This activity is similar to the one described in figure 3-4, checking for inconsistencies related to sources and sinks within a single diagram. The added notation is not actually required for this activity since the inconsistencies are introduced within elements that cross diagram boundaries. However, this activity is required to relate all information contained in a hierarchy of more than two levels.

Chapter Four

CONSISTENCY CHECKER DESIGN AND IMPLEMENTATION

The consistency checker will take a requirements specification in the form of a set of textual structured analysis diagram descriptions, and produce a report of any inconsistencies in inputs, outputs, and controls that occur within the specification. The system will operate similar to other available software tools in the Department of Computer Science at Kansas State University. Input to the system will be obtained from standard input, and the output will be directed to standard output. The system begins by parsing the free-format input, extracting and storing the required input, output, and control information from the diagram descriptions. If an error is encountered while parsing the input, the error and its location within the input will be identified. If the file is successfully parsed, a formatted echo of the input is produced. The parser will not be case sensitive, but when the input is echoed, the diagram description keywords will be in lower-case, with user defined identifiers in upper case. When the echo of input is complete, the consistency checker will link the diagrams into a tree, checking inter-diagram consistency, and then intra-diagram consistency.

CONSISTENCY CHECKER SYSTEM HIERARCHY

System hierarchy is outlined in figure 4-1. The control structure can be broken down into three major portions: DIAGRAM DESCRIPTION PARSER, PRINT DIAGRAM DESCRIPTION, and MAKE CONSISTENCY CHECKS.

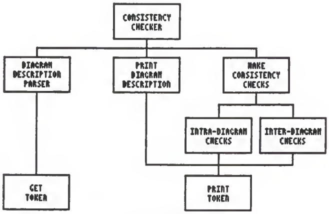


FIGURE 4-1 — SYSTEM HIERARCHY

DIAGRAM DESCRIPTION PARSER will be implemented using a recursive-descent process, making calls to GET TOKEN and PRINT TOKEN. GET TOKEN scans the input stream for tokens, where tokens are delimited by white space, or where appropriate, by commas or colons. PRINT TOKEN changes the internal storage representation of a token into a form suitable for printing.

PRINT DIAGRAM DESCRIPTION will print the free-form input in a more structured form. When printed, all description keywords will be printed in lower case, with all user-defined identifiers in upper case. PRINT DIAGRAM DESCRIPTION also makes calls to PRINT TOKEN.

MAKE CONSISTENCY CHECKS will control the consistency checking activities, making calls to INTER-DIAGRAM CHECKS, and INTRA-DIAGRAM CHECKS. INTER-DIAGRAM CHECKS makes consistency checks of the whole diagram tree, identifying existing inconsistencies in relationships between diagrams. INTRA-DIAGRAM CHECKS makes consistency checks of a single diagram, identifying existing inconsistencies within that diagram. Both INTER-DIAGRAM CHECKS and INTRA-DIAGRAM CHECKS make calls to PRINT TOKEN.

CONSISTENCY CHECKER IMPLEMENTATION

The implementation of the consistency checker will be on a VAX 11/780 computer operating under UNIX, at Kansas State University, Department of Computer Science. The consistency checker will be implemented using Pascal.

The consistency checker will expect input to be directed from standard input, and will direct all output to standard output. The checker can be invoked, recieving all input (terminated by ctrl-D) from the terminal and directing all output to the terminal, by the command

```
check
```

The checker can be invoked, receiving all input from an external file and directing all output to the terminal, by the command

```
check < infile
```

The checker can be invoked, receiving all input from an external file and directing all output to a separate external file, by the command

```
check < infile > outfile
```

No other options are available at invocation. Error messages are outlined, with explanations, in appendix D.

DESCRIPTION OF ALGORITHMS FOR CHECKING CONSISTENCY

Two types of consistency checks must be made: inter-diagram checks and intra-diagram checks. The inter-diagram checks ensure consistency between various levels within the diagram hierarchy. The intra-diagram checks ensure consistency within a single diagram. Each diagram must be checked for consistency. Inter-diagram checking is performed first, followed by intra-diagram checking.

Inter-diagram consistency checking is performed for each activity in the diagram hierarchy that is decomposed in a lower-level detail diagram. The process begins by extracting data sources from the activity and its related detail diagram. The possible data sources are inputs and controls to the activity, and the outputs from the detail diagram. For each source placed in this source list, the extension is used to locate the appropriate data sink. The possible data sinks are inputs and controls to the detail diagram, and outputs from the activity. The extension gives the diagram identifier and data element identifier. If a data element is found in the appropriate position, the data elements are compared. Non-identical data element names signify inconsistencies, which are identified as errors. The location of each identified inconsistency is then printed for the user. After all possible sources have been identified and checked, data element sinks are located. All sinks not previously matched to a data element source are identified as additional inconsistencies, with their locations printed for the user.

Intra-diagram consistency checking is performed for each diagram in the diagram hierarchy. Intra-diagram checks are easier to perform than inter-diagram checks since all required information is located on one diagram. The process begins by extracting data sources. The possible data sources are inputs and controls to the diagram, and the outputs from each activity in the diagram. For each source placed in this source list, the extension is used to locate the appropriate data sink. The possible data sinks are inputs and controls to each activity in the diagram, and outputs from the diagram. Splits/Joins are treated similar to activities. If a data element is found in the appropriate position, the data elements are compared. Non-identical data element names signify inconsistencies, which are identified as errors. The location of each identified inconsistency is then printed for the user. After all possible sources have been identified and checked, data element sinks are located. All sinks not previously matched to a data element source are identified as additional inconsistencies, with their locations printed for the user.

Chapter Five
CONCLUSIONS

Requirements specifications are the basis for developing a system. The requirements specification defines the problem and outlines the characteristics (including constraints) of a correct solution. The requirements specification must answer questions about the system, but an inconsistent specification is unable to do this because the specification contains contradictions. The requirements specification should be analyzable for consistency, with mechanical analysis being advantageous since automated tools can enable easier and more accurate analysis.

Structured analysis diagrams are a graphic system for concisely specifiying requirements of large scale systems. However, the abstraction process that gives structured analysis much of its power also allows inconsistencies in naming information at different abstraction levels. Additional information must be embedded within the structured analysis specification to enable computer tools to ensure consistency. In structured analysis, this embedded information can be in the form of extensions to data element names.

A consistent requirements specification is a necessity when developing a system. The requirements specification lays the groundwork for all subsequent stages. Without a strong foundation, it is unlikely that a correct solution can be completed for a problem. By reducing the number of errors in a system early in the development process, the probability of a correct solution, and a solution with less cost, is increased. Consistency checking of requirements specifications is one method of possibly reducing the number of errors in the implementation of a system, and is therefore beneficial.

References

[Al85] Alford, M. "SREM at the Age of Eight; The Distributed Computing Design System." IEEE Computer (April 1985): pp. 36-46.

[Bo85] Borgida, A., S. Greenspan, and J. Mylopoulos. "Knowledge Representation as the Basis of Requirements Specifications." IEEE Computer (April 1985): pp. 82-91.

[Ca78] Campos, I.M., and G. Estrin. "Concurrent Software System Design Supported by SARA at the Age of One." Proceedings of the 3rd International Conference on Software Engineering (1978): pp. 230-242.

[Co78] Combelic, D. "Experience with SADT." Proceedings of the National Computer Conference (1978): pp. 631-633.

[Gr82] Greenspan, S.J., J. Mylopoulos, and A. Borgida. "Capturing More World Knowledge in the Requirements Specification." Proceedings of the Sixth International Conference on Software Engineering (1982): pp. 225-234.

[Gu84] Gustafson, D.A. "A Requirement Model for the 5th Generation." ACM84 Annual Conference Proceedings (1984): pp. 149-156.

[He83] Heitmeyer, C.L., and J.D. McLean. "Abstract Requirements Specification: A New Approach and Its Application." IEEE Transactions on Software Engineering (SE-9, September 1983): pp. 580-589.

[Ho85] Hoffnagle, G.F., and W.E. Beregi. "Automating the Software Development Process." IBM Systems Journal (Vol. 24, No. 2, 1985): pp. 102-120.

[Hw82] Howden, W.E. "Contemporary Software Development Environments." Communications of the ACM (May 1982): pp. 318-329.

[Rm85] Roman, G.-C. "A Taxonomy of Current Issues in Requirements Engineering." IEEE Computer. (April 1985): pp. 14-22.

[Ro85] Ross, D.T. "Applications and Extensions of SADT." IEEE Computer (April 1985): pp. 25-34.

References

[Ro77a] Ross, D.T. "Reflections on Requirements." IEEE Transactions on Software Engineering (SE-3, January 1977): pp. 2-5.

[Ro77b] Ross, D.T. "Structured Analysis (SA): A Language for Communicating Ideas." IEEE Transactions on Software Engineering (SE-3, January 1977): pp. 16-34.

[Ro77c] Ross, D.T., and K.E. Schoman, Jr. "Structured Analysis for Requirements Definition." IEEE Transactions on Software Engineering (SE-3, January 1977): pp. 6-15.

[Ru79] Ruggiero, W., et al. "Analysis of Data Flow Models Using the SARA Graph Model of Behavior." Proceedings of the National Computer Conference (1979): pp. 975-988.

[Sc85] Scheffer, P.A., A.H. Stone, III, and W.E. Rzepka. "A Case Study of SREM." IEEE Computer (April 1985): pp. 47-54.

[Si85] Sievert, G.E., and T.A. Mizell. "Specification-Based Software Engineering with TAGS." IEEE Computer (April 1985): pp. 56-65.

[Te77] Teichrow, D., and E.A. Hershey, III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." IEEE Transactions on Software Engineering (SE-3, January 1977): pp. 41-48.

[Ya82] Yamamoto, Y., et al. "The Role of Requirements Analysis in the System Life Cycle." Proceedings of the National Computer Conference (1982): pp. 381-387.

Appendix A

B-N-F GRAMMAR FOR STRUCTURED ANALYSIS DIAGRAM DESCRIPTIONS

GENERAL DESCRIPTION

A Structured Analysis Diagram Description contains information related to the contents of a structured analysis diagram. One description is required for each diagram in the system model. When combined, the set of diagram descriptions form the input for the consistency checker. The input can be free-format, with the individual tokens in the description separated by white space, or when indicated in the B-N-F, by commas.

The information in a diagram description is derived from the related diagram. The diagram description contains the diagram name, all inputs, outputs, and controls that cross the diagram boundary, activity information, and split/join information. Activity information includes the activity name, and all inputs, outputs, and controls for the activity. Split information includes the input to the split and the list of outputs from the split. Join information includes the input list to the join and the output from the join.

The amount of information contained within a diagram description is ultimately restricted by the rules governing diagram development (e.g., the number of activities contained within a description has a maximum value of six, since the number of activities within a diagram is limited to six).

SYNTAX DESCRIPTION

```
<dgm_list> ::=
     <dgm_desc> ¦ <dgm_desc> <dgm_list>

<dgm_desc> ::=
     diagram: <activity_id> <dgm_body> end:

<dgm_body> ::=
     <ioc_group> <activity_list> <connect_list>
```

Appendix A

```
<ioc_group> ::=
     <input_list> <control_list> <output_list>

<input_list> ::=
     input: <ioc_list>

<control_list> ::=
     control: <ioc_list>

<output_list> ::=
     output: <ioc_list>

<ioc_list> ::=
     <id_list> ¦ none

<id_list> ::=
     <connect_id> ¦ <connect_id> , <id_list>

<activity_list> ::=
     <activity> ¦ <activity> <activity_list>

<activity> ::=
     activity: <activity_id> <ioc_group>

<connect_list> ::=
     <split_list> <join_list>

<split_list> ::=
     <split> ¦ <split> <split_list> ¦ nil

<split> ::=
     split: <connect_id> <output_list>

<join_list> ::=
     <join> ¦ <join> <join_list> ¦ nil

<join> ::=
     join: <connect_id> <input_list>

<activity_id> ::=
     <id>

<connect_id> ::=
     <id> <extension>
```

Appendix A

```
<id> ::=
     <alpha> <alphanumeric>

<extension> ::=
     . <source> / <sink>

<source> ::=
     <diagram_number> <ioc> <one_to_six> |
     s <one_to_six> | j < one_to_six>

<diagram_number> ::=
     a <one_to_six_list> | a0

<one_to_six> ::=
     1 | 2 | 3 | 4 | 5 | 6

<one_to_six_list> ::=
     <one_to_six> | <one_to_six> <one_to_six_list>

<ioc> ::=
     i | o | c

<alpha> ::=
     a | b | c | ... | x | y | z |
     A | B | C | ... | X | Y | Z | _

<int> ::=
     <digit> | <digit> <int>

<digit> ::=
     0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<alphanumeric> ::=
     <alpha> <alphanumeric> | <digit> <alphanumeric> | nil
```

Appendix B

DIAGRAM DESCRIPTION DEVELOPMENT

A Structured Analysis Diagram Description contains information related to the contents of a structured analysis diagram. One description is required for each diagram in the system model. Before the diagram description can be developed, the diagram must be complete. A complete diagram includes the following:

1) Diagram name,
2) All data elements named,
3) All activities named and numbered,
4) All diagram inputs, controls, and outputs numbered,
5) All activity inputs, controls, and outputs numbered.
6) All data element splits/joins explicitly represented within the diagram.

See [Ro77b] for details on diagram development. Note that item six above is a deviation from current structured analysis styles so is not included in [Ro77b]. A complete diagram is illustrated in figure B-1. The exact syntax for a diagram description can be found in appendix A.

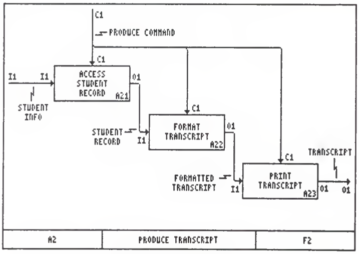


FIGURE B-1 — ORIGINAL DIAGRAM FORM

NUMBERING SPLITS/JOINS

Before data element extensions can be added all splits/joins must be numbered, similar to the numbering of diagram inputs, controls, and outputs. All splits (joins) are numbered arbitrarily beginning with S1 (J1). In figure B-2, C1 has been split twice, giving S1 and S2. Note that two splits are not actually required. Both splits could be combined into one, thus simplifying the diagram description. In the figure, two splits are used to conform to currently accepted diagram style. For data elements being split into new, unique data elements no further additions must be made to the diagram. This also applies to unique data elements being joined into one new, unique data element. For splits and joins where the data element is the same on each branch of the split or join, the name must be copied to each segment of the split or joined data element. For the purpose of reducing diagram clutter, the data element name can simply be placed at the location of the split or join, with the extensions to be placed on individual segments. If this alternative is chosen, it is recommended that only one split or join be present on a data element of this type. See figure B-3.

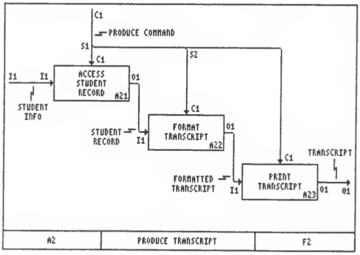


FIGURE B-2 — DIAGRAM WITH LABELED SPLITS

ADDING DATA ELEMENT EXTENSIONS

At this point, all required parts of the diagram should be numbered. It is now possible to begin adding data element extensions.

Data element names are easily made by concatenating the data source identifier and the data sink identifier. The data source and data sink identifiers are made from a combination of the activity number of the source or sink, (i.e., A21) and from the data element number (i.e., I1) of the source or sink. A data source or data sink identifier may also be the number of a split or join (i.e., S1 or J1). The data source and data sink identifiers are concatenated with a slash, and are separated from the data element name with a period. The process can best be shown through example.

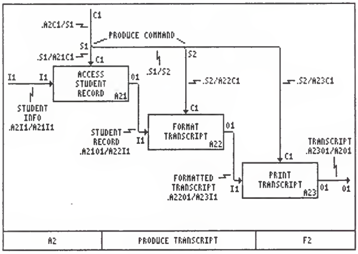


FIGURE B-3 — COMPLETED DIAGRAM

In figure B-3, the extension for the data element STUDENT INFO would be A2I1/A21I1, with the final name becoming STUDENT_INFO.A2I1/A21I1. The name of the data element is STUDENT INFO, therefore, the part before the period becomes STUDENT_INFO (all blanks within the name are replaced by underscores). The first half of the extension is made by combining the activity of the source for the data element, A2, with the data element number, I1. The activity number of the source is the identifier of the diagram, since the data element crosses the diagram boundary, and the data element is input one of the diagram. The second half of the extension is formed similarly from the activiy number of the sink (which is activity A21) and the data element number for that activity (which is I1). Combining these two parts gives the final extension of A2I1/A21I1. This is then concatenated with the first part to give the result, STUDENT_INFO.A2I1/A21I1.

As a second example, the final data element name for the control to activity A21, ACCESS STUDENT RECORD, becomes PRODUCE_COMMAND.S1/A21C1. The first half is produced by replacing all blanks within the original data element name with underscores. The second half is produced by combining the data element source and sink identifiers. The source identifier is S1, since it comes from split one. The sink identifier is A21C1, since it goes to activity-two-one, and is control-one to that activity. Combining the data element source and sink identifiers with a slash, and concatenating the original data element name and the extension with a period, the final data element name becomes PRODUCE_COMMAND.S1/A21C1.

The remaining data element names with their appropriate extensions are given in figure B-3. Remember that activity numbers are taken from different places depending on whether the data element crosses a diagram boundary, and whether the activity number refers to the data element source or sink. Sources that are inputs or controls to the diagram get the activity number from the diagram itself. All other sources are outputs from activities (or splits/joins) within the diagram and get the activity number directly from that activity (or directly from the split/join). Sinks that are outputs from the diagram get the activity number from the diagram itself. All other sinks are inputs to activities (or splits/joins) within the diagram, or are controls to activities within the diagram. These sinks get the activity number directly from the activity (or directly from the split/join).

Appendix C

DIAGRAM DESCRIPTION EXAMPLE FROM CHAPTER ONE

```
diagram:          STUDENT_DATABASE

  input:          STUDENT_INFO.A0I1/S1
  control:        CREATE_COMMAND.A0C1/A1C1,
                  PRODUCE_COMMAND.A0C2/A2C1,
                  MODIFY_COMMAND.A0C3/A3C1
  output:         CREATE_MSG.A1O1/A0O1,
                  TRANSCRIPT.A2O1/A0O2,
                  MODIFY_MSG.A3O1/A0O3

  activity:       CREATE_STUDENT

    input:        STUDENT_INFO.S1/A1I1
    control:      CREATE_COMMAND.A0C1/A1C1
    output:       CREATE_MSG.A1O1/A0O1

  activity:       PRODUCE_TRANSCRIPT

    input:        STUDENT_INFO.S1/A2I1
    control:      PRODUCE_COMMAND.A0C2/A2C1
    output:       TRANSCRIPT.A2O1/A0O2

  activity:       MODIFY_TRANSCRIPT

    input:        STUDENT_INFO.S1/A3I1
    control:      MODIFY_COMMAND.A0C3/A3C1
    output:       MODIFY_MSG.A3O1/A0O3

  split:          STUDENT_INFO.A0I1/S1

    output:       STUDENT_INFO.S1/A1I1,
                  STUDENT_INFO.S1/A2I1,
                  STUDENT_INFO.S1/A3I1

end:
```

Appendix C

```
diagram:         MODIFY_TRANSCRIPT

  input:         STUDENT_INFO.A3I1/S1
  control:       MODIFY_COMMAND.A3C1/S2
  output:        MODIFY_MSG.A32O1/A3O1

  activity:      ACCESS_STUDENT_RECORD

    input:       STUDENT_INFO.S1/A31I1
    control:     MODIFY_COMMAND.S2/A31C1
    output:      STUDENT_RECORD.A31O1/A32I1

  activity:      UPDATE_STUDENT_RECORD

    input:       STUDENT_RECORD.A31O1/A32I1,
                 STUDENT_INFO.S1/A32I2
    control:     MODIFY_COMMAND.S2/A32C1
    output:      MODIFY_MSG.A32O1/A3O1,
                 UPDATED_STUDENT_RECORD.A32O1/A33I1

  activity:      STORE_STUDENT_RECORD

    input:       UPDATED_STUDENT_RECORD.A32O1/A33I1
    control:     MODIFY_COMMAND.S2/A33C1
    output:      NONE

  split:         STUDENT_INFO.A3I1/S1

    output:      STUDENT_INFO.S1/A31I1,
                 STUDENT_INFO.S1/A32I2

  split:         MODIFY_COMMAND.A3C1/S2

    output:      MODIFY_COMMAND.S2/A31C1,
                 MODIFY_COMMAND.S2/A32C1,
                 MODIFY_COMMAND.S2/A33C1

end:
```

Appendix C

```
diagram:          CREATE_STUDENT

  input:          STUDENT_ID.A1I1/A11I1
  control:        CREATE_COMMAND.A1C1/S1
  output:         CREATE_MSG.A11O2/A1O1

  activity:       CREATE_STUDENT_RECORD

    input:        STUDENT_ID.A1I1/A11I1
    control:      CREATE_COMMAND.S1/A11C1
    output:       STUDENT_RECORD.A11O1/A12I1,
                  CREATE_MSG.A11O2/A1O1

  activity:       STORE_STUDENT_RECORD

    input:        STUDENT_RECORD.A11O1/A12I1
    control:      CREATE_COMMAND.S1/A12C1
    output:       NONE

  split:          CREATE_COMMAND.A1C1/S1

    output:       CREATE_COMMAND.S1/A11C1,
                  CREATE_COMMAND.S1/A12C1

end:
```

Appendix C

```
diagram:          PRODUCE_TRANSCRIPT

  input:          STUDENT_INFO.A2I1/A21I1
  control:        PRODUCE_COMMAND.A2C1/S1
  output:         TRANSCRIPT.A23O1/A2O1

  activity:       ACCESS_RECORD

    input:        STUDENT_INFO.A2I1/A21I1
    control:      PRODUCE_COMMAND.S1/A21C1
    output:       STUDENT_RECORD.A21O1/A22I1

  activity:       FORMAT_TRANSCRIPT

    input:        STUDENT_RECORD.A21O1/A22I1
    control:      PRODUCE_COMMAND.S1/A22C1
    output:       FORMATTED_TRANSCRIPT.A22O1/A23I1

  activity:       PRINT_TRANSCRIPT

    input:        FORMATTED_TRANSCRIPT.A22O1/A23I1
    control:      PRODUCE_COMMAND.S1/A23C1
    output:       TRANSCRIPT.A23O1/A2O1

  split:          PRODUCE_COMMAND.A2C1/S1

    output:       PRODUCE_COMMAND.S1/A21C1,
                  PRODUCE_COMMAND.S1/A22C1,
                  PRODUCE_COMMAND.S1/A23C1

end:
```

Appendix D

ERROR MESSAGES

1) ERROR: "diagram:" expected

   Scanning error. Expecting the keyword "diagram:" as input.

2) ERROR: "end:" expected

   Scanning error. Expecting the keyword "end:" as input.

3) ERROR: "input:" expected

   Scanning error. Expecting the keyword "input:" as input.

4) ERROR: "output:" expected

   Scanning error. Expecting the keyword "output:" as input.

5) ERROR: "activity:" expected

   Scanning error. Expecting the keyword "activity:" as input.

6) ERROR: "control:" expected

   Scanning error. Expecting the keyword "control:" as input.

7) ERROR: unexpected comma or keyword

   Scanning error. Expecting identifier, but found comma or keyword.

8) ERROR: no A0 diagram

   Scanning error. A diagram description for the highest abstraction level was not encountered in the input file.

9) ERROR: multiple A0 diagrams

Scanning error. More than one diagram description for the highest abstraction level was encountered in the input file.

10) ERROR: unmatched source(s) in diagram

Consistency error. One or more source data elements were not matched to an appropriate sink data element.

11) ERROR: unmatched sink(s) in diagram

Consistency error. One or more sink data elements were not matched to an appropriate source data element.

Appendix E

IMPLEMENTATION SOURCE CODE

```
program check (input,output);

(*********************************************************
  Check -- Program to check consistency of a structured
    analysis diagram description.  The required format for
    the input file is described in Master's Thesis:

    CONSISTENCY CHECKING OF REQUIREMENTS SPECIFICATIONS
            USING STRUCTURED ANALYSIS DIAGRAMS

                            by

                      Aaron Friesen

  Program Completed December 1986.

  Note:  All input is taken from Standard Input, and all
    output is directed to Standard Output.
 *********************************************************)

type
  str80  = packed array[1..80] of char;
  dgmptr = ^dgmrec;                   (* diagram info        *)
  iocptr = ^iocrec;                   (* input, output,
                                                control info *)
  iocrec = record
             name : str80;            (* ioc name            *)
             next : iocptr;           (* next ioc            *)
           end; (* record *)
  actptr = ^actrec;                   (* activity info       *)
  actrec = record
             name   : str80;          (* activity name       *)
             ins    : iocptr;         (* input list          *)
             ctrls  : iocptr;         (* control list        *)
             outs   : iocptr;         (* output list         *)
             detail : dgmptr;         (* detail diagram
                                                for activity *)
             next   : actptr;         (* next activity       *)
           end; (* record *)
  sjptr = ^sjrec;                     (* split, join info    *)
  sjrec = record
            whole : str80;            (* sj name             *)
            parts : iocptr;           (* input/output list   *)
            next  : sjptr;            (* next sj             *)
```

```
          end; (* record *)
{ dgmptr = ^dgmrec; }                     (* diagram info          *)
  dgmrec = record
              name       : str80;    (* diagram name          *)
              hasparent  : boolean;  (* parent flag           *)
              ins        : iocptr;   (* input list            *)
              ctrls      : iocptr;   (* control list          *)
              outs       : iocptr;   (* output list           *)
              acts       : actptr;   (* activity list         *)
              splits     : sjptr;    (* split list            *)
              joins      : sjptr;    (* join list             *)
              next       : dgmptr;   (* next diagram          *)
           end; (* record *)
  chkptr = ^chkrec;
  chkrec = record                         (* check list info       *)
              name1 : str80;              (* source/sink name      *)
              name2 : str80;
              kind  : char;               (* source/sink kind      *)
              next  : chkptr;             (* next source/sink
                                                       to check    *)
           end; (* record *)

var
  cccc : char;                            (* global, next
                                                 char in input  *)
  dgm : dgmptr;                           (* head pointer
                                                 for diagrams   *)
  word : str80;                           (* temp input
                                                 variable       *)
  line : integer;                         (* current line of
                                                    input file  *)
  error : boolean;                        (* global error flag     *)

procedure GetWord (var word:str80; var line:integer);

(**********************************************************
  GetWord -- Scans input stream for one 'word.'  A 'word'
    is defined as anything delimited by white space.
    Also, a comma or colon is assumed to mark the end of
    a 'word.'

  word -- 'word' being input
  line -- current line number within input file being
    scanned
  cccc -- global variable of next input character to be
    used
 **********************************************************)
```

```
var
  length : integer;  (* length of input word *)

  function GetCh : char;

  (********************************************************
    GetCh -- Gets a character from the input stream,
      converting all white space to blanks.

    cccc -- input character
  ********************************************************)

  var ch : char;  (* temporary input character *)

  begin
    if eoln then line := line + 1;
    read(ch);
    if ch in ['a'..'z']
      then ch := chr(ord(ch) - 32)
      else if (ch = tab) or (ch = chr(12)) then ch := ' ';
    GetCh := ch;
  end;

begin
  word := '  ';
  length := 0;
  while not eof and (cccc = ' ') do cccc := GetCh;
  while not eof and (cccc <> ':')
         and (cccc <> ' ') and (cccc <> ',') do
    begin
      length := length + 1;
      word[length] := cccc;
      cccc := GetCh;
    end; (* while *)
  if cccc = ':'
    then
      begin
        word[length+1] := cccc;
        cccc := GetCh;
      end (* then *)
    else if (length = 0) and (cccc = ',')
           then
             begin
               word := ', ';
               cccc := GetCh;
             end; (* then *)
```

```
end; (* GetWord *)

procedure PrintWord (word:str80; ln:boolean);

(**********************************************************
  PrintWord -- Print a word, dropping all trailing blanks.

  word -- 'word' being printed
  ln -- boolean indicating new line (as in write vs.
    writeln)
 **********************************************************)

var i : integer;

begin
  i := 1;
  while (word[i] <> ' ') and (i <= 80) do
    begin
      write(word[i]);
      i := i + 1;
    end; (* while *)
  if ln then writeln;
end; (* PrintWord *)

procedure err (num:integer);

(**********************************************************
  err -- Print an error message

  num -- message number
 **********************************************************)

begin
  write('ERROR: ');
  case num of
     1 : writeln('"diagram:" expected');
     2 : writeln('"end:" expected');
     3 : writeln('unexpected comma or keyword');
     4 : writeln('"input:" expected');
     5 : writeln('"output:" expected');
     6 : writeln('"activity:" expected');
     7 : writeln('"control:" expected');
     8 : writeln('no A0 diagram');
     9 : writeln('multiple A0 diagrams');
    10 : write('unmatched source(s) in diagram ');
    11 : write('unmatched sink(s) in diagram ');
  end; (* case *)
  error := true;
end; (* err *)

function KeyWord (word:str80) : boolean;

(**********************************************************
  KeyWord -- determine if the word is a keyword

  word -- word to be checked
 **********************************************************)

begin
  KeyWord := (word = 'DIAGRAM:')  or (word = 'INPUT:')   or
             (word = 'OUTPUT:')   or (word = 'CONTROL:') or
             (word = 'ACTIVITY:') or (word = 'SPLIT:')   or
             (word = 'JOIN:')     or (word = 'END:')     or
             (word = ', ');
end; (* KeyWord *)

procedure GetDgm (var dgm:dgmptr; var word:str80;
                  var line:integer);

(**********************************************************
  GetDgm -- Get a diagram from the input stream

  dgm -- diagram pointer
  word -- next word to be used from input
  line -- current line number of input file
 **********************************************************)

label 9999;

  procedure GetIOC (var ioc:iocptr; var word:str80;
                    var line:integer);

(**********************************************************
    GetIOC -- get an IOC portion from the input stream

    ioc -- ioc pointer
    word -- next word to be used from input
    line -- current line number of input file
 **********************************************************)

  begin
    if KeyWord(word)
      then err(3)
      else
```

```
        begin
          new(ioc);
          with ioc^ do
            begin
              name := word;
              next := nil;
            end; (* while *)
          GetWord(word, line);
          if word = ', '
            then
              begin
                GetWord(word, line);
                GetIOC(ioc^.next, word, line);
              end; (* then *)
        end; (* else *)
  end; (* GetIOC *)

  procedure GetBox (var act:actptr; var word:str80;
                    var line:integer);

(**********************************************************
   GetBox -- get a box portion from the input stream

   box -- box pointer
   word -- next word to be used from input
   line -- current line number of input file
 **********************************************************)

  label 9999;

  begin
    if KeyWord(word)
      then err(3)
      else
        begin
          new(act);
          with act^ do
            begin
              name := word;
              ins := nil;
              ctrls := nil;
              outs := nil;
              detail := nil;
              next := nil;
            end; (* with *)
          GetWord(word, line);
          if word = 'INPUT:'
          then
            begin
              GetWord(word, line);
              GetIOC(act^.ins, word, line);
              if error then goto 9999;
              if word = 'CONTROL:'
                then
                  begin
                    GetWord(word, line);
                    GetIOC(act^.ctrls, word, line);
                    if error then goto 9999;
                    if word = 'OUTPUT:'
                      then
                        begin
                          GetWord(word, line);
                          GetIOC(act^.outs, word, line);
                          if error then goto 9999;
                        end (* then *)
                      else err(5);
                  end (* then *)
                else err(7);
            end (* then *)
          else err(4);
        if not error and (word = 'ACTIVITY:')
          then
            begin
              GetWord(word, line);
              GetBox(act^.next, word, line);
            end; (* then *)
      end; (* else *)
  9999:
  end; (* GetBox *)

  procedure GetSplit (var split:sjptr; var word:str80;
                      var line:integer);

(***********************************************************
    GetSplit -- get a split portion from the input stream

    split -- split pointer
    word -- next word to be used from input
    line -- current line number of input file
***********************************************************)

  begin
    if KeyWord(word)
      then err(3)
```

```
     else
       begin
         new(split);
         with split^ do
           begin
             whole := word;
             parts := nil;
             next := nil;
           end; (* with *)
         GetWord(word, line);
         if word = 'OUTPUT:'
           then
             begin
               GetWord(word, line);
               GetIOC(split^.parts, word, line);
             end (* then *)
           else err(5);
       end; (* else *)
    if not error and (word = 'SPLIT:')
      then
        begin
          GetWord(word, line);
          GetSplit(split^.next, word, line);
        end; (* then *)
  end; (* GetSplit *)

  procedure GetJoin (var join:sjptr; var word:str80;
                     var line:integer);

(*********************************************************
    GetJoin -- get a join portion from the input stream

    join -- join pointer
    word -- next word to be used from input
    line -- current line number of input file
*********************************************************)

  begin
    if KeyWord(word)
      then err(3)
      else
        begin
          new(join);
          with join^ do
            begin
              whole := word;
              parts := nil;
              next := nil;
            end; (* with *)
          GetWord(word, line);
          if word = 'INPUT:'
            then
              begin
                GetWord(word, line);
                GetIOC(join^.parts, word, line);
              end (* then *)
            else err(5);
        end; (* else *)
    if not error and (word = 'JOIN:')
      then
        begin
          GetWord(word, line);
          GetJoin(join^.next, word, line);
        end; (* then *)
  end; (* GetJoin *)

begin
  if KeyWord(word)
    then err(3)
    else
      begin
        new(dgm);
        with dgm^ do
          begin
            name := word;
            hasparent := false;
            ins := nil;
            ctrls := nil;
            outs := nil;
            acts := nil;
            splits := nil;
            joins := nil;
            next := nil;
          end; (* with *)
        GetWord(word, line);
        if word = 'INPUT:'
          then
            begin
              GetWord(word, line);
              GetIOC(dgm^.ins, word, line);
              if error then goto 9999;
              if word = 'CONTROL:'
                then
                  begin
                  GetWord(word, line);
                  GetIOC(dgm^.ctrls, word, line);
                  if error then goto 9999;
                  if word = 'OUTPUT:'
                    then
                      begin
                        GetWord(word, line);
                        GetIOC(dgm^.outs, word, line);
                        if error then goto 9999;
                        if word = 'ACTIVITY:'
                          then
                            begin
                              GetWord(word, line);
                              GetBox(dgm^.acts, word,
                                     line);
                              if error then goto 9999;
                              if word = 'SPLIT:'
                                then
                                  begin
                                    GetWord(word, line);
                                    GetSplit(dgm^.splits,
                                             word, line);
                                  end; (* then *)
                              if error then goto 9999;
                              if word = 'JOIN:'
                                then
                                  begin
                                    GetWord(word, line);
                                    GetJoin(dgm^.joins,
                                            word, line);
                                  end; (* then *)
                              if word = 'END:'
                                then GetWord(word, line)
                                else err(2);
                            end (* then *)
                          else err(6);
                      end (* then *)
                    else err(5);
                end (* then *)
              else err(7);
          end (* then *)
        else err(4);
    end; (* else *)
if word = 'DIAGRAM:'
  then
    begin
      GetWord(word, line);
```

```
        GetDgm(dgm^.next, word, line);
      end (* then *)
    else if (word <> '  ') and not error then err(1);
9999 :
end; (* GetDgm *)

procedure PrintDgms (dgm : dgmptr);

(**********************************************************
  PrintDgms -- print the diagrams from the input file

  dgm -- diagram pointer
 **********************************************************)

  procedure PrintIOC (ioc:iocptr);

(**********************************************************
    PrintIOC -- print the IOC portion of the diagram

    IOC -- IOC pointer

 **********************************************************)

  begin
      while ioc <> nil do
        begin
          PrintWord(ioc^.name,false);
          ioc := ioc^.next;
          if ioc = nil
            then writeln
            else
              begin
                writeln(',');
                write(tab, tab);
              end; (* else *)
        end; (* while *)
  end; (* PrintIOC *)

  procedure PrintBox (act:actptr);

(**********************************************************
    PrintBox -- print the Box portion of the diagram

    box -- box pointer
 **********************************************************)

  begin
```

```
    while act <> nil do
      begin
        writeln;
        write('  activity:', tab);
        PrintWord(act^.name,true);
        writeln;
        write('    input:', tab);
        PrintIOC(act^.ins);
        write('    control:', tab);
        PrintIOC(act^.ctrls);
        write('    output:', tab);
        PrintIOC(act^.outs);
        act := act^.next;
      end; (* while *)
  end; (* PrintBox *)

  procedure PrintSJ (sj:sjptr; info1, info2:str80);

(**********************************************************
    PrintSJ -- print the SJ portion of the diagram

    sj -- sj pointer
 **********************************************************)

  begin
    while sj <> nil do
      begin
        writeln;
        write('  ');
        PrintWord(info1,false);
        write(tab);
        PrintWord(sj^.whole,true);
        writeln;
        write('    ');
        PrintWord(info2,false);
        write(tab);
        PrintIOC(sj^.parts);
        sj := sj^.next;
      end; (* while *)
  end; (* PrintSJ *)

begin
  while dgm <> nil do
    begin
      writeln;
      write('diagram:', tab);
      PrintWord(dgm^.name,true);
      writeln;
      write('  input:', tab);
      PrintIOC(dgm^.ins);
      write('  control:', tab);
      PrintIOC(dgm^.ctrls);
      write('  output:', tab);
      PrintIOC(dgm^.outs);
      PrintBox(dgm^.acts);
      PrintSJ(dgm^.splits,'split: ','output: ');
      PrintSJ(dgm^.joins,'join: ','input: ');
      writeln;
      writeln('end:');
      writeln;
      dgm := dgm^.next;
    end; (* while *)
end; (* PrintDgm *)

procedure CheckConsistency (dgm:dgmptr);

(**********************************************************
  CheckConsistency -- Check consistency of the diagrams

  dgm -- diagram pointer
 **********************************************************)

var
  d, parent : dgmptr;
  a : actptr;
  count : integer;

  procedure MakeDgmLink (dgm:dgmptr; act:actptr);

  (**********************************************************
    MakeDgmLink -- link the diagrams into a tree structure

    dgm -- diagram pointer
    act -- activity pointer
**********************************************************)

  begin
    while (dgm^.next <> nil)
            and (act^.name <> dgm^.name) do
      dgm := dgm^.next;
    if act^.name = dgm^.name
      then
        begin
          act^.detail := dgm;
          dgm^.hasparent := true;
        end; (* then *)
  end; (* MakeDgmLink *)

  procedure Append (name:str80; kind:char;
                    var list:chkptr);

(*********************************************************
    Append -- append the input name to the list

    name -- input name to be appended
    kind -- kind of name being added to the list
    list -- list of names
*********************************************************)

  begin
    new(list^.next);
    list := list^.next;
    list^.next := nil;
    list^.name1 := name;
    list^.name2 := name;
    list^.kind := kind;
  end; (* Append *)

  procedure MakeList (ioc:iocptr; kind:char;
                      var list:chkptr);

(*********************************************************
    MakeList -- build the list of source/sink information

    ioc -- IOC pointer
    kind -- kind of item being added to the list
    list -- list of sources/sinks
*********************************************************)

  begin
    while ioc <> nil do
      begin
        if ioc^.name <> 'NONE'
          then Append(ioc^.name,kind,list);
        ioc := ioc^.next;
      end; (* while *)
  end; (* MakeList *)

  function FindAndDelete (name:str80;
                          var list:chkptr):boolean;
```

```
var
  current, back : chkptr;
  found : boolean;

begin
  found := false;
  back := list;
  current := list^.next;
  while (current <> nil) and not found do
    if current^.name2 = name
      then
        begin
          back^.next := current^.next;
          dispose(current);
          current := nil;
          found := true;
        end (* then *)
      else
        begin
          back := current;
          current := current^.next;
        end; (* else *)
  FindAndDelete := found;
end; (* FindAndDelete *)

procedure PrintList (list:chkptr);

(*********************************************************
   PrintList -- Print the names of the items contained
     within the input list.

   list -- list to be printed
 *********************************************************)

 begin
   if list <> nil
     then
       begin
         case list^.kind of
           'i' : write(tab, '(input)   ');
           'o' : write(tab, '(output)  ');
           'c' : write(tab, '(control) ');
           's' : write(tab, '(split)   ');
           'j' : write(tab, '(join)    ');
         end; (* case *)
         PrintWord(list^.name1,true);
         PrintList(list^.next);
        end; (* then *)
  end; (* PrintList *)

  procedure CheckWithin (dgm:dgmptr);

(*********************************************************
    CheckWithin -- Check the consistency of information
      that is within a diagram description.

    dgm -- diagram pointer
*********************************************************)

  var
    sources, sinks : chkptr;
    tempsource, tempsink : chkptr;
    back : chkptr;
    a : actptr;
    sj : sjptr;

  begin
    if dgm <> nil
      then
        begin
          new(sources);
          sources^.next := nil;
          new(sinks);
          sinks^.next := nil;
          tempsource := sources;
          tempsink := sinks;
          MakeList(dgm^.ins,'i',tempsource);
          MakeList(dgm^.ctrls,'c',tempsource);
          MakeList(dgm^.outs,'o',tempsink);
          a := dgm^.acts;
          while a <> nil do
            begin
              MakeList(a^.ins,'i',tempsink);
              MakeList(a^.ctrls,'c',tempsink);
              MakeList(a^.outs,'o',tempsource);
              a := a^.next;
            end; (* while *)
          sj := dgm^.splits;
          while sj <> nil do
            begin
              Append(sj^.whole,'s',tempsink);
              MakeList(sj^.parts,'s',tempsource);
              sj := sj^.next;
            end; (* while *)
          sj := dgm^.joins;
          while sj <> nil do
            begin
              Append(sj^.whole,'j',tempsource);
              MakeList(sj^.parts,'j',tempsink);
              sj := sj^.next;
            end; (* while *)
          back := sources;
          tempsource := sources^.next;
          while tempsource <> nil do
            begin
              if FindAndDelete(tempsource^.name2,sinks)
                then
                  begin
                    back^.next := tempsource^.next;
                    dispose(tempsource);
                    tempsource := back;
                  end; (* then *)
              back := tempsource;
              tempsource := tempsource^.next;
            end;
          if sources^.next <> nil
            then
              begin
                err(10);
                PrintWord(dgm^.name,true);
                PrintList(sources^.next);
              end; (* then *)
          if sinks^.next <> nil
            then
              begin
                err(11);
                PrintWord(dgm^.name,true);
                PrintList(sinks^.next);
              end; (* then *)
          a := dgm^.acts;
          while a <> nil do
            begin
              CheckWithin(a^.detail);
              a := a^.next;
            end; (* while *)
        end; (* then *)
  end; (* CheckWithin *)

  procedure CheckBoundary (dgm:dgmptr);

(**********************************************************
    CheckBoundary -- Check the consistency of information
      that crosses diagram boundaries.

    dgm -- diagram pointer
**********************************************************)

  var a : actptr;

    procedure RemoveLast (wordin:str80; var wordout:str80);

(**********************************************************
      RemoveLast -- Removes the last half of the extension
        on an IOC item

      wordin -- the original form of the IOC to be updated
      wordout -- the IOC in its updated form
 *********************************************************)

    var
      i, j : integer;

    begin
      i := 80;
      while (wordin[i] <> '/') and (i > 1) do
        i := i - 1;
      if wordin[i] = '/'
        then
          for j := i to 80 do wordin[j] := ' ';
      wordout := wordin;
    end; (* RemoveLast *)

    procedure RemoveFirst (wordin:str80; var
wordout:str80);

(**********************************************************
      RemoveFirst -- Removes the first half of the
        extension on an IOC item

      wordin -- the original form of the IOC to be updated
      wordout -- the IOC in its updated form
**********************************************************)

    var
      i, j, k, n : integer;

    begin
      i := 80;
      while (wordin[i] = ' ') and (i > 1) do
        i := i - 1;
      if wordin[i+1] = ' '
        then
          begin
            j := i - 1;
            while (wordin[j] <> '/') and (j > 1) do
              j := j - 1;
            if wordin[j] = '/'
              then
                begin
                  k := j - 1;
                  while (wordin[k] <> '.') and (k > 1) do
                    k := k - 1;
                  if wordin[k] = '.'
                    then
                      begin
                        for n := 1 to i - j do
                          wordin[k+n] := wordin[j+n];
                        for n := k + i - j + 1 to i do
                          wordin[n] := ' ';
                      end; (* then *)
                end; (* then *)
          end; (* then *)
      wordout := wordin;
    end; (* RemoveFirst *)

    procedure MakeCheck (act:actptr; parent:dgmptr);

(*********************************************************
      MakeCheck -- Performs the actual consistency check
        of the information that crosses diagram boundaries.

      act -- activity pointer
      parent -- diagram pointer pointing to the parent
        diagram
*********************************************************)

    var
      sources, sinks : chkptr;
      tempsource, tempsink : chkptr;
      back : chkptr;

      procedure AddemSources (ioc:iocptr; kind:char;
                              var list:chkptr);

 (*********************************************************
        AddemSources -- Build a source list from the
          detail diagram.  Before the sources are added to
          the list, remove the first half of the extension.

        ioc -- ioc pointer
        kind -- kind of item being added to the list
        list -- list of sources
 ********************************************************)

      begin
        while ioc <> nil do
          begin
            if ioc^.name <> 'NONE'
              then
                begin
                  Append(ioc^.name,kind,list);
                  case kind of
                    'i' : RemoveFirst(list^.name1,
                                      list^.name2);
                    'o' : RemoveLast(list^.name1,
                                      list^.name2);
                    'c' : RemoveFirst(list^.name1,
                                       list^.name2);
                  end; (* case *)
               end; (* then *)
            ioc := ioc^.next;
          end; (* while *)
      end; (* AddemSources *)

      procedure AddemSinks (ioc:iocptr; kind:char;
                            var list:chkptr);

(********************************************************
        AddemSinks -- Build a sink list from the detail
          diagram.   Before the sinks are added to the
          list, remove the last half of the extension.

        ioc -- ioc pointer
        kind -- kind of item being added to the list
        list -- list of sinks
 ********************************************************)

      begin
        while ioc <> nil do
          begin
            if ioc^.name <> 'NONE'
              then
              begin
                Append(ioc^.name,kind,list);
                case kind of
                  'i' : RemoveLast(list^.name1,
                                   list^.name2);
                  'o' : RemoveFirst(list^.name1,
                                   list^.name2);
                  'c' : RemoveLast(list^.name1,
                                   list^.name2);
                end; (* case *)
              end; (* then *)
          ioc := ioc^.next;
        end; (* while *)
    end; (* AddemSinks *)

  begin
    if act^.detail <> nil
      then
        begin
          new(sources);
          sources^.next := nil;
          new(sinks);
          sinks^.next := nil;
          tempsource := sources;
          tempsink := sinks;
          AddemSources(act^.ins,'i',tempsource);
          AddemSources(act^.ctrls,'c',tempsource);
          AddemSources(act^.outs,'o',tempsource);
          AddemSinks(act^.detail^.ins,'i',tempsink);
          AddemSinks(act^.detail^.ctrls,'c',tempsink);
          AddemSinks(act^.detail^.outs,'o',tempsink);
          back := sources;
          tempsource := sources^.next;
          while tempsource <> nil do
            begin
              if FindAndDelete(tempsource^.name2,sinks)
                then
                  begin
                    back^.next := tempsource^.next;
                    dispose(tempsource);
                    tempsource := back;
                  end; (* then *)
              back := tempsource;
              tempsource := tempsource^.next;
            end; (* where *)
          if sources^.next <> nil
            then
```

```
                  begin
                    err(10);
                    PrintWord(parent^.name,false);
                    write(' in activity ');
                    PrintWord(act^.name,true);
                    PrintList(sources^.next);
                  end; (* then *)
              if sinks^.next <> nil
                then
                  begin
                    err(11);
                    PrintWord(act^.name,true);
                    PrintList(sinks^.next);
                  end; (* then *)
            end; (* then *)
      end; (* MakeCheck *)

    begin
      if dgm <> nil
        then
          begin
            a := dgm^.acts;
            while a <> nil do
              begin
                CheckBoundary(a^.detail);
                MakeCheck(a,dgm);
                a := a^.next;
              end; (* while *)
          end; (* then *)
    end; (* CheckBoundary *)

  begin
    d := dgm;
    while d <> nil do
      begin
        a := d^.acts;
        while a <> nil do
          begin
            MakeDgmLink(dgm,a);
            a := a^.next;
          end; (* while *)
        d := d^.next;
      end; (* while *)
    count := 0;
    d := dgm;
    while d <> nil do
      begin
      if not d^.hasparent
        then
          begin
            count := count + 1;
            parent := d;
          end; (* then *)
      d := d^.next;
    end; (* while *)
  if count = 0
    then err(8)
    else if count > 1
           then
             begin
               err(9);
               d := dgm;
               while d <> nil do
                 begin
                   if not d^.hasparent
                     then
                       begin
                         write('  diagram:  ');
                         PrintWord(d^.name,true);
                       end; (* then *)
                   d := d^.next;
                 end; (* while *)
             end (* then *)
           else
             begin
               writeln('Beginning Intra-diagram
                                    Consistency Check');
               CheckWithin(parent);
               writeln('Completed Intra-diagram
                                    Consistency Check');
               writeln('Beginning Inter-diagram
                                    Consistency Check');
               CheckBoundary(parent);
               writeln('Completed Inter-diagram
                                    Consistency Check');
             end; (* else *)
end; (* CheckConsistency *)

begin  (* main *)
  error := false;
  cccc := ' ';
  dgm := nil;
  line := 1;
  writeln;
  writeln('Beginning Input Scan');
  if not eof
    then
      begin
        GetWord(word, line);
        if word = 'DIAGRAM:'
          then
            begin
              GetWord(word, line);
              GetDgm(dgm, word, line);
            end (* then *)
          else err(1);
      end; (* then *)
  if error
    then
      begin
        write('Processing Stopped At "');
        PrintWord(word,false);
        writeln('" Near Line ', line:2);
      end (* then *)
    else
      begin
        writeln('Completed Input Scan');
        writeln('Beginning Echo Of Input');
        PrintDgms(dgm);
        writeln('Completed Echo Of Input');
        CheckConsistency(dgm);
        writeln('Completed Consistency Check');
        if not error then writeln('No Errors Encountered');
      end; (* else *)
end. (* main *)
```

CONSISTENCY CHECKING OF REQUIREMENTS SPECIFICATIONS USING STRUCTURED ANALYSIS DIAGRAMS

by

AARON NOBLE FRIESEN

B.S., Kansas State University, 1985

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

ABSTRACT

Requirements specifications are the basis for developing a system, defining the problem and outlining the characteristics (including constraints) of a correct solution. The requirements specification must answer questions about the system, but an inconsistent specification is unable to do this because the specification contains contradictions. The requirements specification should be analyzable for consistency. Automated tools enable easier and more accurate analysis.

Structured analysis diagrams are a system for concisely specifiying requirements of large scale systems, yet inconsistencies are possible in naming information at different abstraction levels. Extensions to data element names showing the element's source and sink enable computer tools to insure consistency.

A consistent requirements specification is a necessity when developing a system. Consistency checking of requirements specifications is one method of possibly reducing the number of errors in the implementation of a system, and is therefore beneficial. By reducing the number of errors in a system early in the development process, the probability of a correct solution, and a solution with less cost, is increased.