ModelMaker: A Tool for Rapid Modeling from Device Descriptions

By
Andreas Indra Gunawan
gunawan@vt.edu

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering
(Computer Engineering Option)

W.R. Cyre, Chairman
J.R. Armstrong
F.G. Gray

20 May 1998
Blacksburg, Virginia

Keywords: Tool, ModelMaker, VHDL, Specification

ModelMaker: A Tool for Rapid Modeling from Device Descriptions

Andreas I. Gunawan

(ABSTRACT)

This thesis describes a tool that facilitates rapid modeling of devices from informal documents. The ModelMaker tool facilitates the construction of models by analyzing the source specification document and presenting it based on the modeler's need. ModelMaker analyzes and indexes the source document for the noun phrases and identifiers it contains. When the modeler specifies the name of a pin or device that the modeler is working on, ModelMaker will recover any behavioral or structural information about the particular pin or device. ModelMaker can return this information based on the order of how relevant the information is to the model that the modeler is trying to build. The modeling language that can be used for this tool is VHDL. The initial VHDL model is derived from a block diagram of the source document using a schematic capture tool. This VHDL model can be edited inside ModelMaker to add behavioral code and to insert source document fragments as comments.

Acknowledgements

I would like to thank Dr. Walling R. Cyre, my academic and research advisor, for all of his assistance and support that he has given to me. I would also like to thank Dr. James R. Armstrong and Dr. F. Gail Gray for serving on my committees and for all of their assistance and support for this project.

I would also like to thank Ritesh Sojitra and Jeff Hess, graduate research assistants under Dr. Walling R. Cyre, for any help that they have provided to me throughout this project.

And Finally, I would like to thank my parents, sisters, brothers in law, friends and colleagues for their continuous support and encouragement.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.1 Motivation

In today's technology, device modeling has become increasingly important. Simulation and synthesis are two applications of device modeling. Languages such as VHDL and Verilog are becoming more popular among device engineers. Engineers, also called modelers, use device modeling to test or rebuild legacy systems, to build libraries of commercial components, or even to build new devices. This device-modeling concept, however, comes with a price. Especially in dealing with legacy systems, a considerable amount of expertise and effort is needed because most of these systems come with ambiguous and incomplete specification documentation.

The ModelMaker tool facilitates the modeling of these devices by analyzing the specification document and presenting it based on the modeler's need. When the modeler specifies the name of a pin or device that the modeler is working on, ModelMaker will recover any behavioral or structural information about the particular pin or device (behavioral and structural descriptions are defined in the Structured Logic Design with VHDL [2]). ModelMaker can return this information based on the order of how relevant the information is to the model that the modeler is trying to build.

The only input needed by ModelMaker is a text version of the specification document, and a basic model file for the device. In the current implementation, VHDL is the only modeling language that can be read by ModelMaker. The basic VHDL file can be generated automatically using a schematic capture tool such as Synopsys Graphical Environment (SGE).

ModelMaker is a prototype application. It should not be used as a final application. During the duration of this particular project, three main versions of ModelMaker have been released for prototype demonstration. The first version of

ModelMaker is capable of analyzing a source specification document and storing it in an hierarchical document object. This version also loads the VHDL file, and parses the VHDL identifiers. However, it only provides search capability on identifiers. The second version of ModelMaker includes the Noun Phrase Extractor component. This version is able to parse a ModelMaker document object to extract all of the noun phrases that occur in the document. Finally, ModelMakerIII is the third version of ModelMaker. This version involved a total remodeling of its graphical user interface. This version is designed to have more options and controls that comply with the Windows standards. Another major addition in ModelMakerIII is the Template Constructor component. This version is able to communicate with the template-constructor, which is written in Prolog, and present the results in the ModelMaker Source Window in a template format.

This thesis is an extension of the paper presented at the IVC/VIUF 98 [10].

1.2 Overview

ModelMaker is an executable application that is targeted for the Windows 95 platform. Its main working window is a dialog box that contains two edit-windows and some controls. The two edit-windows are the Source Window and the Model Window. The layout of this main working window is displayed in Figure 1-1.

The Source Window, placed in the top portion of the main working window, is where the analyzed specification document elements will be displayed by ModelMaker. This read-only window allows the user to select and copy some or all of the text and paste it into the Model Window. The Model Window, placed in the bottom portion of the main working window, is initialized with the basic VHDL model. This window is editable, and the text can be saved back to the original model-text file. The primary objective is to allow the modeler to view source document information and the Model Window at the same time.

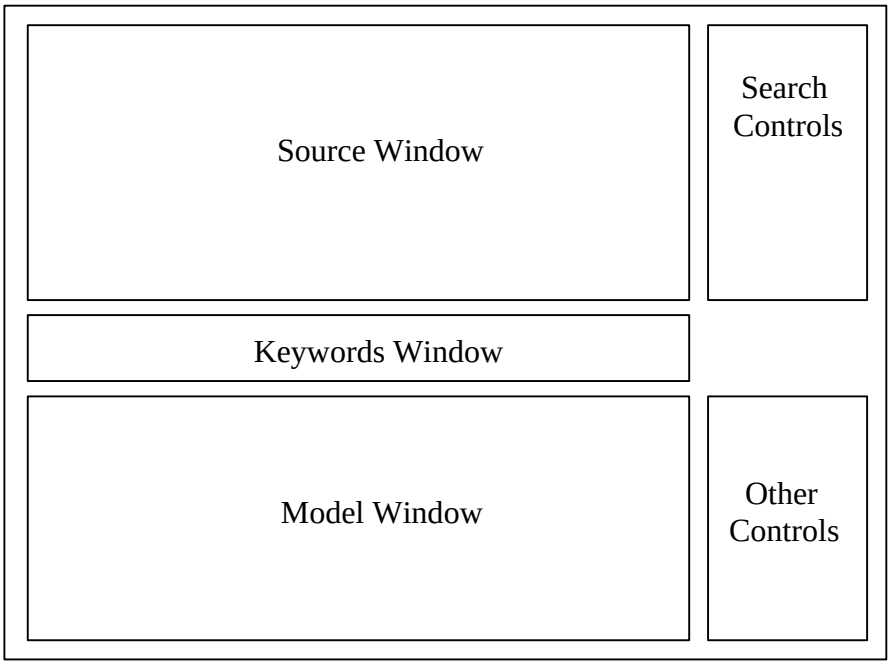


Figure 1-1: ModelMaker Interface Layout

ModelMaker requires two main input files, the document-specification text file and the basic VHDL code file. The modeler should be able to obtain these two input files directly from the device documentation. Device documentation is assumed to consist of a behavioral and structural description of the device, a high-level block diagram, and possibly a pin description table and other tables. A scanner with an Optical Character Recognition (OCR) program should be able to recover most of the text. This ASCII text file forms the source document file that is needed by ModelMaker. Some formatting issues on this text file are discussed in section 4.2.4.2. The high-level block diagram is equally important to the document-specification text file. By redrawing this diagram into a schematic-capture-application that is capable of generating a VHDL file, the modeler can use this VHDL file as the framework for his/her modeling using ModelMaker. While ModelMaker loads this VHDL file, it parses all of the I/O pin identifiers, the component names, and other identifiers. Similarly, while ModelMaker loads the document-specification text file, it parses all of the noun phrases. This information is used by the ModelMaker tool to create the identifier and the noun phrase search indexes on the specification document. Since the search indexes are constructed during the initialization

of ModelMaker, searches are performed very rapidly. This encourages the modeler to browse around the search phrases without any annoying time penalty.

The keywords window, located between the Source Window and the Model Window, contains a drop-down list box and radio buttons. The drop-down list box is where the list of the identifiers or the list of the noun phrases will be presented to the modeler. The modeler can either choose one of the keywords listed there, or type his/her own phrase in the keywords window. This keyword is used by the ModelMaker tool as the search phrase. ModelMaker will then search the specification document for behavioral and structural information about the keywords. The radio control button lets the modeler choose whether to display the identifier keyword list or the noun phrase keyword list. The identifier and the noun phrase lists are obtained from the identifier search index and the noun phrase search index respectively.

In modeling a device, a modeler focuses on a single pin or component of the device. When given this pin name or component name as the keywords, ModelMaker will show the information about the particular pin or component in the Source Window. The modeler can read the information, copy selections to the Model Window, or use the information to complete the model being built. The latest implementation of the ModelMaker's main window is shown in Figure 1-2.

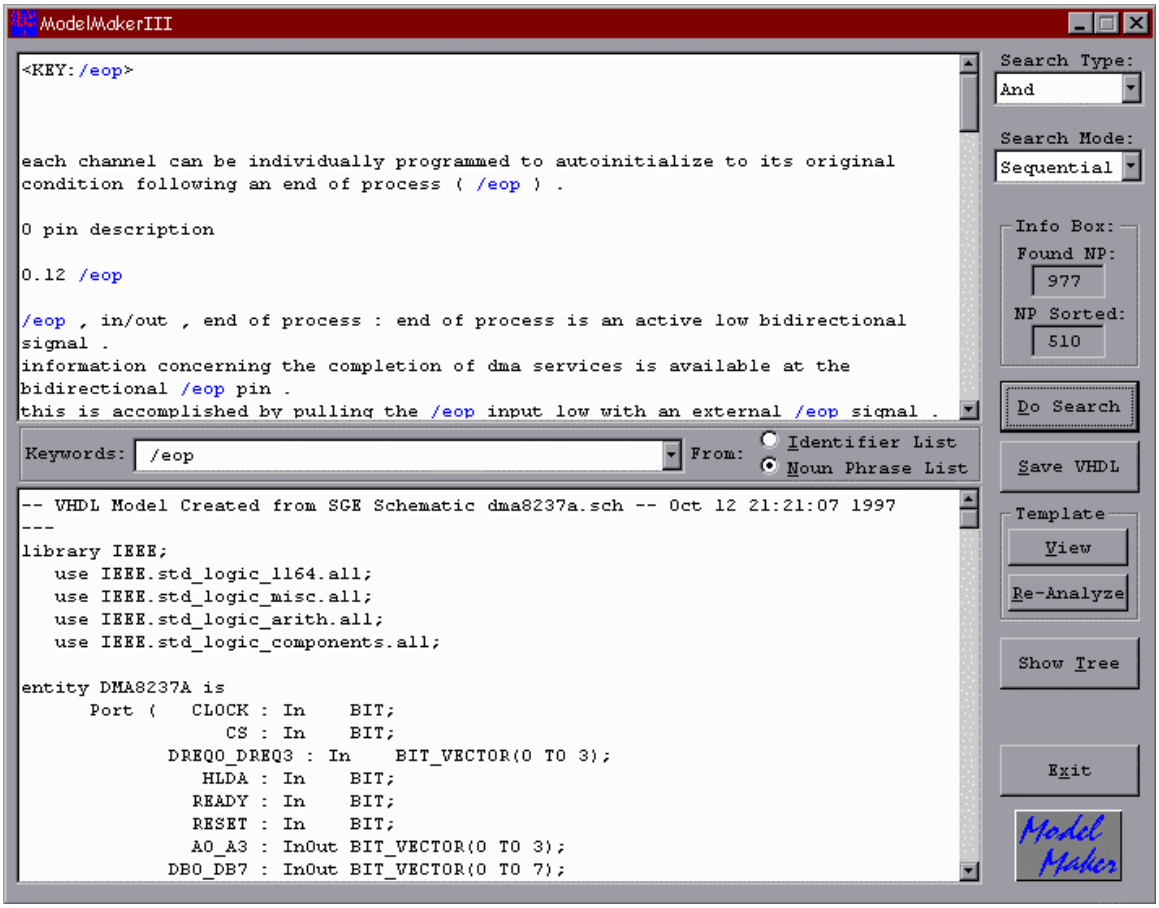


Figure 1-2: Captured ModelMaker Main Window

The search control section includes the search-type and the search-mode drop-down lists. The search-type drop-down list deals with how ModelMaker combines the search results of multi-word identifiers. The possible options given in this search-type drop-down list are the Boolean “and” and “or.” For the search-mode, the possible options given are “sequential,” “relevance,” “behavioral,” and “structural.” More details on these search modes and types are discussed in section 4.4, Keyword Searches.

The rest of the controls are simple buttons. In the current implementation of ModelMaker, there are a total of seven control buttons. They are the “Do-Search” button, the “Save-VHDL” button, the “Template-View” button, the “Template-Re-

Analyze" button, the "Show Tree" button, the "Exit" button, and finally the "ModelMaker" button.

The modeler needs to press the "Do-Search" button every time he/she wants to view the information on a new keyword phrase. When this button is pressed, ModelMaker will show the information on the selected search key in the Source Window. Once the modeler is done with the modeling, he/she can save the new VHDL code by pressing the "Save-VHDL" button. Pressing this button will overwrite the old VHDL file with the new VHDL code shown in the Model Window.

The "Template-View" button and the "Template-Re-Analyze" button deal with the external program, Template Constructor. The Template Constructor is a set of Prolog rules that will analyze sentences using a set of predefined templates. As the analyzing process can take a substantial amount of time, ModelMaker can use a pre-analyzed result. This pre-analyzed result can be saved in the ModelMaker directory with a file name "_out.txt." By pressing the "Template-View" button, ModelMaker will read this pre-analyzed result, and present it in the Source Window. If the modeler does want to re-analyze the document, he/she can use the "Template-Re-Analyze" button. When this button is pressed, ModelMaker will invoke the Template Constructor, and present the output in the Source Window.

ModelMaker also gives the option to the modeler to view the document parse trees. This can be done by pressing the "Show-Tree" button. This capability is used mainly for debugging purposes. By viewing the tree, one can see how the sentences are parsed within ModelMaker. As the process of constructing these trees also requires a great amount of time, a copy of preprocessed trees is stored in the ModelMaker local directory. By default, pressing this button will cause ModelMaker to read from the local file. If the modeler wanted to reconstruct the parse trees, he/she can select this option during ModelMaker initialization.

The last button, the “ModelMaker” button, functions as the “about” button. By pressing this button, the modeler can see the version and copyright information about this ModelMaker tool.

To illustrate the functionality of ModelMaker, the search results for the identifier keyword “hrq” are presented below. For this example, the specification document used is the commercial 8237A-DMA-controller specification-document by Intel. These searches were done with the sequential mode search (Figure 1-3) and the behavioral mode search (Figure 1-4). In the current implementation, ModelMaker reads all of the characters in lowercase. Therefore, search results displayed in the Source Window will all be in lowercase letters. From these two different searches, with the same keyword but different mode, one can see how the results are ordered. The mode selections primarily deal with how to order the search results.

<KEY:hrq>

0 pin description

0.15 hrq

hrq , out , hold request : this is the hold request to the cpu and is used to request control of the system bus .
dreq causes 8237a to issue the hrq .

2 dma operation

2.2 active cycle

when the 8237a is in the idle cycle and a non-masked channel requests a dma service , the device will output an hrq to the microprocessor and enter the active cycle .

2.2.1 single transfer mode

if dreq is held active throughout the single transfer , hrq will go inactive and release the bus to the system .

2.2.4 cascade mode

the hrq and hlda signals from the additional 8237a are connected to the dreq and dack signals of a channel of the initial 8237a .
the 8237a will respond to dreq and dack but all other outputs except hrq will be disabled .

2.3 transfer types

2.3.3 priority

after completion of a service , hrq will go inactive and the 8237a will wait for hlda to go low before activating hrq to service another channel .

4 programming

the 8237a will accept programming from the host processor any time that hlda is inactive ; this is true even if hrq is active .

5 application information (note 1)

the multimode dma controller issues a hrq to the processor whenever there is at least one valid dma request from a peripheral device .

Figure 1-3: Sequential Mode Result

<KEY:hrq>

the hrq and hlda signals from the additional 8237a are connected to the dreq and dack signals of a channel of the initial 8237a .
the 8237a will respond to dreq and dack but all other outputs except hrq will be disabled .

the multimode dma controller issues a hrq to the processor whenever there is at least one valid dma request from a peripheral device .

the 8237a will accept programming from the host processor any time that hlda is inactive ; this is true even if hrq is active .

if dreq is held active throughout the single transfer , hrq will go inactive and release the bus to the system .

when the 8237a is in the idle cycle and a non-masked channel requests a dma service , the device will output an hrq to the microprocessor and enter the active cycle .

hrq , out , hold request : this is the hold request to the cpu and is used to request control of the system bus .
dreq causes 8237a to issue the hrq .

after completion of a service , hrq will go inactive and the 8237a will wait for hlda to go low before activating hrq to service another channel .

Figure 1-4: Behavioral Mode Result

1.3 Research Contributions

The idea of ModelMaker was initiated by Dr. Walling R. Cyre. The main idea was to combine existing tools, a Parser and the SemAnal from the ASPIN project [12], to build a better and more useful tool for device modelers. The research contributions that were made by the author of this thesis to the ModelMaker tool are listed below.

- **ModelMaker I**
  - *Main C++ classes for the data structure of ModelMaker.*

    The primary C++ classes that are used for the data structure of ModelMaker were built intentionally for ModelMaker. The document element hierarchy (sections, paragraphs, and sentences) was built using object oriented

programming. The program files that are used to define these classes are the "Specification.cpp" and the "Specification.h" files.

- *The Lexer.*

  A Lexer class is built to decompose input text into tokens. Lexer differentiates between a period in the middle of a number and a period at the end of a sentence. The tokenizing rules used in Lexer are given in section 4.2.4.2.

- *Improved word-indexing algorithm for faster performance.*

  The CDictionary class for the ASPIN Parser was used as a template in building the CDictionary class for ModelMaker. Due to poor performance of the look-up algorithm used in this CDictionary class, a new algorithm was designed. In the new version, a hash table of pointers is used as an index into the CDictionary object. This new algorithm made a significant improvement in the search performance of ModelMaker. More information about the hash table is presented in section 4.3.1.3.

- *Specification document reader algorithm.*

  An algorithm was built to transform the specification document text into the ModelMaker document data structure. The first version of this algorithm required additional markup to be added to the specification document. This was to ensure that ModelMaker read the document hierarchy correctly. The markup style used was similar to the Hypertext Markup Language (HTML). The current version does not need this additional markup anymore. A new algorithm was built to read the document specification file with standard formatting. However, the style of the documents that ModelMaker is able to read automatically is still limited by some rules. These rules are explained with more details in section 4.2.4.2. These current rules are able to recognize the test documents with a small amount of adjustment in the specification document text formatting.

- *VHDL Identifiers Extractor algorithm.*

  A VHDL reader algorithm was created to read the VHDL input file into the ModelMaker tool. This algorithm is capable of extracting most of the

identifiers listed in the VHDL file. An identifier is defined as a signal name, a port name, a component name, or an entity name. ModelMaker presents this list of identifiers to the modeler as search keywords.

- *Search capabilities for the identifiers.*

  The ability to search the ModelMaker specification document data structure based on the identifiers is implemented in this version. During its initialization process, ModelMaker searches its document data structure for any occurrence of the identifiers to build a search index. Using this index, ModelMaker presents search results in its Source Window. This implementation also allows the modeler to perform Boolean "And" and "Or" searches for multi-word identifiers. To add more flexibility, ModelMaker allows the modeler to select any token in the Model Window as a keyword phrase. If any word in the new keyword phrase is not found in the search index, ModelMaker will perform a new search on this new keyword at run-time. The result will then be added to the search index for future reference.

- *Sorting capabilities for the search results.*

  The ability to sort the search results based on predefined modes is implemented in the ModelMaker tool. The predefined modes are the *sequential mode*, the *relevance mode*, the *behavioral mode*, and the *structural mode*. Using this capability, the modeler can view the search results in the order of how "behavioral" the information is or how "structural" the information is. Each of these modes is discussed with more details in the algorithm section (see section 4.4).

- *Graphical user interface improvements.*

  The initial layout of the ModelMaker interface was a simple dialog-based application with no functional control attached to it. The release version of ModelMaker uses this dialog-based application as its template. The layout was modified, and more control buttons were added. Most of the contributions made in this version are on the actual engines behind the controls.

- **ModelMaker II**
  - *Integrated Noun Phrase Extractor.*

    This version of ModelMaker incorporates the modified ASPIN Parser, now called the Noun Phrase Extractor. Using this ability, ModelMaker can extract most of the noun phrases that occur in a specification document. Similarly to the identifiers, these noun phrases are then used as search indexes.
  - *Search capabilities for noun phrases.*

    In this version, the ability to search the specification document data structure based on the noun phrase list was implemented. From the noun phrases found by the Noun Phrase Extractor, ModelMaker searches its specification document for any occurrence of these noun phrases during its initialization. Similarly to the identifier case, these search results are also placed in the search index.

- **ModelMaker III**
  - *Sentence Extractor.*

    The ability to extract sentence parse trees from the ModelMaker document-specification data structure was implemented in this version. This ability is needed by ModelMaker to be able to generate input files for the Template Constructor. These sentence parse trees are developed by the Noun Phrase Extractor. Each of these sentences is then converted to a Prolog format.
  - *Integrated template-constructor.*

    This version of ModelMaker integrates the modified ASPIN Semantic Analyzer (SemAnal), now called the Template Constructor, into the second version of ModelMaker. Using this capability, ModelMaker tries to match the structure of the sentences, from the Noun Phrase Extractor parse-tree, to the pre-defined templates.

- *Source and Model Window Modification.*

  New Source and Model Windows are implemented in this ModelMaker version. These windows are based on the CRichEditCtrl class of the Microsoft Foundation Class library, version 4.21. The text in these windows can be formatted with bold, italic, and colors. This is done to allow ModelMaker to show the search results in a better format. Some special markup rules were made to implement this formatting capability. The Template Constructor attaches these markup tags into its output to control the formatting of the templates. The new Source and Model Window support the standard editing tools such as *cut*, *copy*, and *paste*.

- *Graphical user interface total redesigned.*

  The graphical user interface of this ModelMaker version was redesigned. This was done to clean up the source code and to satisfy Windows programming standards. Moreover, this improves ModelMaker portability. To use ModelMaker tools in a platform other than Windows 95, all that the programmer needs to do is to redesign the interface in the new platform. Some adjustments in the platform-dependent parts of the code had to be added to the code during this process. However, that should not cause too many problems, as the main engine of ModelMaker is written in ANSI C++ code. This ANSI C++ code can be ported to a Unix machine or to any other machines, that has the ANSI C++ compiler.

- *Pre-analyzed results, and pre-captured parse trees.*

  As the template constructing process may take a very long period, this version of ModelMaker allows the modeler to save and re-access templates. This implementation can save a lot of time for the modeler. For each new specification document text, the modeler is only required to run the Template Constructor once. By saving the result of the Template Constructor locally, ModelMaker can re-use this result for its next run. The same applies to the parse trees structure. ModelMaker can use the pre-computed parse trees, instead of regenerating the trees each time it is initialized.

- **Noun Phrase Extractor**
  - *Rebuilt classes and memory-leak check ability.*

    Some of the classes for the Noun Phrase Extractor were rebuilt. This was done to ease the communication between ModelMaker and the Noun Phrase Extractor. Another reason was to improve the error handling mechanism, and to add protection to the data structure of the classes. A memory leak checker was also added in each class. This checker is only active during the debug mode.
  - *Multi-word recognition.*

    The current algorithm used in the Noun Phrase Extractor has the capability to recognize multi-word tokens in a sentence. This ability allows the Noun Phrase Extractor to understand multi-word tokens such as "consist of," "greater than," or even "greater than or equal to." More discussion about this multi-word recognition is presented in section 5.1.2.
  - *Rule-skipping capability.*

    The ability to skip one or more words in using the grammar rules was implemented in the Noun Phrase Extractor. This ability allows the Noun Phrase Extractor to recognize more sentences. However, due to slow performance, this capability is not utilized in the current ModelMaker.
  - *Noun Phrase Extractor algorithm.*

    A new function was developed to perform the noun phrase extraction from a chart built by the ASPIN Parser. This function scans through all of the constituents of the existing chart and extracts all of the noun phrase constituents out of it. This is the function that is used by the ModelMaker tool to extract all of the noun phrases out of the specification document object.

- **Template Constructor**
  - *Quintus to SWI Prolog rules conversion.*

    Since the original Template Constructor code, the ASPIN Semantic Analyzer, was written in Quintus Prolog (Unix based application), some adjustments were needed to convert the Prolog rules to SWI Prolog (Windows 95-based application).
  - *Template formatting.*

    Some Prolog rules were added to the Template Constructor to define the template formatting for ModelMaker.

- **Test Case**
  - *DMA 8237A controller specification document.*

    A DMA-8237A controller specification document [15] was used as a test case. This DMA 8237A specification document was scanned and converted automatically to a text format using commercial OCR software. Two schematic drawings were made from the drawings inside the specification documents. These schematics are the structural and the behavioral views of the device. Two VHDL files were then generated from these schematics.

The current ModelMaker, ModelMakerIII, contains twenty-nine C++ classes written in about 8000 lines of code. The ModelMaker is compiled for the Windows 95 platform.

CHAPTER 2: RELEVANT WORK

Electronic Design Automation (EDA) is used extensively by hardware designers. Many educational institutions and commercial companies provide a variety of EDA tools, which reduce design time, and increase productivity. The electronic format of designs allows designers to edit, share, or re-use their designs easily. Moreover, synthesis tools help designers to examine, test, and change their designs. Virginia Tech (VPI&SU) has designed a number of EDA tools for research purposes. Some of these tools are the ASPIN systems [12], VHDLCad [5], and Modeler's Assistant [3,16,19]. Commercial companies such as i-Logix Inc., Synopsys Inc., and Cadence Design System Inc. are just a few of many EDA tool providers. Hewlett-Packard maintains, in their web site[1], a long list of currently available commercial EDA tools. Of these many EDA tools, none of them provide natural language analysis capability. Instead, many of these EDA tools start their design cycle from state diagrams or schematic diagrams. Some of the other tools let the modelers to start from the hardware description language (HDL) level directly, as some modelers prefer to create their own models. The primary objective of this chapter is to discuss some of these currently available EDA applications and to justify the need of ModelMaker.

The Automated Specifications Interpreter (ASPIN) project is a research project being conducted by the Automatic Design Research Group (ADRG) at Virginia Tech. The objective of this research project is to develop applications that improve designer productivity. ASPIN accomplishes this objective by helping a user to develop good specifications and helping the designers understand the information behind the specification documents. The two components of the ASPIN system of interest here are the Parser and the Semantic Analyzer. For more information about the Automatic Design Research Group, contact adrg@vt.edu.

The Parser component is an application that constructs parse trees for English sentences. For any given string of tokens (words, numbers and punctuation) the Parser generates zero or more parse trees. Some of the trees may represent sentences. Others represent lesser constituents such as noun phrases. For example, Figure 2-1 shows a parse tree for the sentence "the sci uses standard nrz format" and for one of its noun phrases. The grammar used by the parser was developed in a study of the English used to describe microprocessors [1].

```
s(2,c(
  s1,[c(
    ss2,[c(
      n1,[c(
        np12,[t(det,the),c(
          head2,[t(id,sci)])])],c(
            pred2,[c(
              avs2,[t(verb,uses)]),c(
                n1,[c(
                  np19,[c(
                    adjs1,[t(adj,standard)]),c(
                      head4,[t(id,nrz),c(
                        head1,[t(noun,format)])])])])])],t(.,.)])).
```

Figure 2-1: Semantic Analyzer Input

Using the result of the Parser, a modeler can use the ASPIN Semantic Analyzer [12] to analyze the information that is carried by the sentences. The Semantic Analyzer attempts to extract the category of the sentence and represent it as a conceptual graph [8,9,20]. To illustrate this process, Figure 2-2 shows the conceptual graph formed by the Semantic Analyzer for the sentence "the sci uses standard nrz format." The input to the Semantic Analyzer was the parse tree of Figure 2-1.

[1] Current location of the list is at http://hpcc923.external.hp.com/csocat/eda/eda/maineda.htm. This URL can be changed at any time by the Hewlett-Packard web-master. Contact the Hewlett-Packard web-master for more information.

```
the sci uses standard nrz format .

[ 1 : use : use ]
        ->( gindx : 1 )
        ->( agnt : agnt ) -> [ 2 ]
        ->( obj : obj ) -> [ 3 ]
[ 2 : id : sci ]
        ->( det : the )
[ 3 : format : format ]
        ->( gindx : 1 )
        ->( name : null ) -> [ 4 ]
        ->( attr : adj ) -> [ 5 ]
[ 4 : id : nrz ]
[ 5 : standard : standard ]
        ->( gindx : 2 )
```

Figure 2-2: Semantic Analyzer Output

By analyzing this Semantic Analyzer result, a modeler who is familiar with the conceptual graphs could read the conceptual graph instead of the main sentence in building his/her model. However, the process of preparing the sentence manually and running both the Parser and the Semantic Analyzer manually can be time consuming and may be irritating. ModelMaker incorporates the Parser and the Semantic Analyzer to extract knowledge that will help the modeler produce models from English descriptions more rapidly.

VHDLCad and Modeler's Assistant, also developed at Virginia Tech (VPI&SU) as research projects, are two other EDA tools to reduce design time for hardware designers. These two applications capture information in the form of process model graphs (PMGs). These tools generate simple VHDL code from the PMGs. These tools serve the same purpose with the ModelMaker tool, help designers to reduce the design time. These tools also use the same modeling target as the ModelMaker tool, the VHDL language. However, neither of these tools has the natural language analyzer capability. This shows the need of the ModelMaker tool.

No commercial tool[2] begins its design cycle directly with English text. Most EDA tools use schematic diagrams, state diagrams, or data flow diagrams for design capture. Moreover, some EDA tools simply let the modeler to start their modeling process from the HDL level. The STATEMATE [13,14] tool by i-Logix Inc., for example, uses state charts, activity charts and module charts to capture the modeler's design. More information on the state charts and activity charts can be found in the Specification and Design of Embedded Systems [11]. The schematic-design-editor subject is discussed in the Structured Logic Design with VHDL [2]. Once a design is captured and validated, STATEMATE can generate VHDL code from it.

Similarly, Synopsys Graphical Environment Shell [21] and Concept-HDL [6] allow modelers to design their devices in an initial stage before actually creating the VHDL or Verilog code. In this case, however, the initial stage is the schematic diagram. All of these applications, again, do help the designers in speeding up their design time. Yet, they also lack any natural language analyzer capabilities.

Compared to all of these other electronic design automation tools, ModelMaker does have natural language analyzer capabilities. It combines the two ASPIN natural language analyzers, the Parser and the Semantic Analyzer, in creating a better design environment. ModelMaker also provides the search capability to allow modelers to work on any particular part of the device. Moreover, ModelMaker allows the modeler to view both the information and the VHDL code at the same time. This is important for a rapid modeling process.

[2] The lists of the currently available EDA tools are obtained from the HP web-site and from the *EE-Times* web-site. The latter can be found at http://techweb.cmp.com/eet/eda/edalinks.html. As in the case with the list provided by HP, this URL can be changed at any time. Consult the *EE-Times* web-master for more information.

CHAPTER 3: MODELING WITH MODELMAKER

This chapter describes the operation of ModelMaker. An example is used here to show how a modeler can construct a device model using ModelMaker. At the same time, a description of how the internal classes are managed (created, deleted, and used) by ModelMaker is also presented in this chapter. The primary objective of this chapter is to present an overall view of how ModelMaker works and to show how ModelMaker can be used by hardware modelers.

The product data sheet for the Intel 8237A direct memory access controller chip [15] is used as an example source document. In creating this model, the modeler is assumed to have access to a scanner, Optical-Character-Recognition (OCR) software, and the Synopsys Graphical Environment (SGE) shell software. The scanner and software are needed to prepare the two input files needed by the ModelMaker tool, the specification document text (source file) and the initial VHDL model file.

The first task for the modeler is to scan the specification document. The first page of the scanned Intel DMA 8237A specification document is reproduced in Figure 3-1. Once this is done, the OCR software captures the text of the scanned document. Some part of the OCR result is shown in Figure 3-2 for an illustration. For more information on the input files, refer to section 4.2.4.

8237A
HIGH PERFORMANCE
PROGRAMMABLE DMA CONTROLLER
(8237A, 8237A-4, 8237A-5)

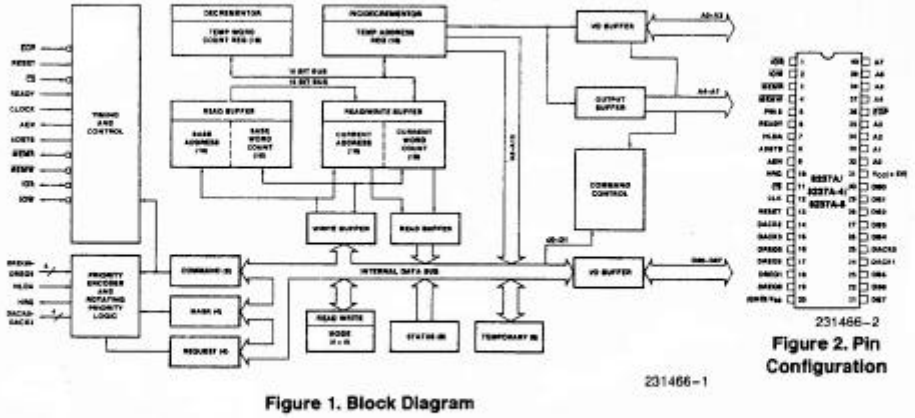
- **Enable/Disable Control of Individual DMA Requests**
- **Four Independent DMA Channels**
- **Independent Autoinitialization of All Channels**
- **Memory-to-Memory Transfers**
- **Memory Block Initialization**
- **Address Increment or Decrement**
- **High Performance: Transfers up to 1.6M Bytes/Second with 5 MHz 8237A-5**
- **Directly Expandable to Any Number of Channels**
- **End of Process Input for Terminating Transfers**
- **Software DMA Requests**
- **Independent Polarity Control for DREQ and DACK Signals**
- **Available in EXPRESS —Standard Temperature Range**
- **Available in 40-Lead Cerdip and Plastic Packages**

(See Packaging Spec, Order # 231369)

The 8237A Multimode Direct Memory Access (DMA) Controller is a peripheral interface circuit for microprocessor systems. It is designed to improve system performance by allowing external devices to directly transfer information from the system memory. Memory-to-memory transfer capability is also provided. The 8237A offers a wide variety of programmable control features to enhance data throughput and system optimization and to allow dynamic reconfiguration under program control.

The 8237A is designed to be used in conjunction with an external 8-bit address latch. It contains four independent channels and may be expanded to any number of channels by cascading additional controller chips. The three basic transfer modes allow programmability of the types of DMA service by the user. Each channel can be individually programmed to Autoinitialize to its original condition following an End of Process (EOP). Each channel has a full 64K address and word count capability.

The 8273A-4 and 8237A-5 are 4 MHz and 5 MHz versions of the standard 3 MHz 8237A respectively.



231466-1

Figure 1. Block Diagram

231466-2

Figure 2. Pin Configuration

October 1989
Order Number: 231466-004

Figure 3-1: Scanned Intel DMA 8237A Controller Specification Document (1st Page)

```
8237A HIGH PERFORMANCE PROGRAMMABLE DMA CONTROLLER (8237A, 8237A-4, 8237A-5).

Features:
* Enable/Disable Control of Individual DMA Requests
* Four Independent DMA Channels
* Independent Autoinitialization of All Channels
* Memory-to-Memory Transfers
* Memory Block Initialization
* Address Increment or Decrement
* High Performance: Transfers up to 1.6M Bytes/Second with 5 MHz 8237A-5
* Directly Expandable to Any Number of Channels
* End of Process Input for Terminating Transfers
* Software DMA Requests
* Independent Polarity Control for DREQ and DACK Signals
* Available In EXPRESS (Standard Temperature Range)
* Available In 40-Lead Cerdip and Plastic Packages
  (See Packaging Spec, Order #231369)

The 8237A Multimode Direct Memory Access (DMA) Controller is a peripheral
interface circuit for microprocessor systems.  It is designed to improve
system performance by allowing external devices to directly transfer
information from the system memory.  Memory-to-memory transfer capability
is also provided.  The 8237A offers a wide variety of programmable control
features to enhance data throughput and system optimization and to allow
dynamic reconfiguration under program control.

The 8237A is designed to be used in conjunction with an external 8-bit
address latch.  It contains four independent channels and may be expanded
to any number of channels by cascading additional controller chips.  The
three basic transfer modes allow programmability of the types of DMA
service by the user.  Each channel can be individually programmed to
Autoinitialize to its original condition following an End of Process (/EOP).
Each channel has a full 64K address and word count capability.

The 8273A-4 and 8237A-5 are 4 MHz and 5 MHz versions of the standard 3 MHz
8237A respectively.
…
…
…
```

Figure 3-2: Partial OCR Result of the Scanned Document

As described in section 4.2.4.2, the result from the OCR software may require some format adjustment to make it fully readable by the ModelMaker tool. Depending on the quality of the document, the result may also need to be checked by spell-checking software. The adjustment and checking need to be done manually before this source document file is given to ModelMaker. ModelMaker attempts to automatically identify the hierarchical structure of a given source document. Sending a source document with a format that does not follow the rules defined in section 4.2.4.2 to ModelMaker may cause some misidentification of the source document hierarchy. Consequently, ModelMaker may not perform at its best.

The current version of ModelMaker (v. 3.08) contains lexical knowledge of about 4000 words in its standard dictionary. This dictionary was constructed manually from words accumulated throughout the ASPIN project and the ModelMaker project. Although this dictionary is adequate for research purposes, it is far from being complete. Consequently, it is suggested that each of the new source documents be scanned for unknown words. During the ModelMaker project, a utility program was created to automatically extract all unknown words out of source document files. The task of adding these words and their respective lexical categories, however, is still manual.

The second manual task for the modeler is to generate the initial VHDL model file. Depending on whether the modeler is interested in building a behavioral or structural model of the device, he/she needs to enter the appropriate schematic to the schematic capture software. For behavioral modeling, only the device outline with external pins is used. For structural modeling, a structural decomposition (block diagram) is used. As a warning, the components of block diagrams are not usually discussed in the source specification document, so the structural information actually complicates the modeling task. The behavioral schematic diagram, the structural schematic diagram, or even both of them can usually be found in the device specification document. In the specification document shown in Figure 3-1, the structural schematic is shown in the bottom left corner (block diagram section) and the behavioral pin schematic diagram is shown in the bottom right corner (pin configuration section). Both the behavioral and the structural schematic of this source specification document are drawn using the Synopsys Graphical Environment (SGE) shell, and two VHDL models were extracted from them. These schematics and VHDL models are presented in Appendix B and Appendix C of this thesis.

For illustration, small behavioral and structural schematics were developed using SGE shell, and are shown in Figure 3-3 and Figure 3-4. The initial VHDL model files generated from these schematics are presented in Figure 3-5 and Figure 3-6, respectively.

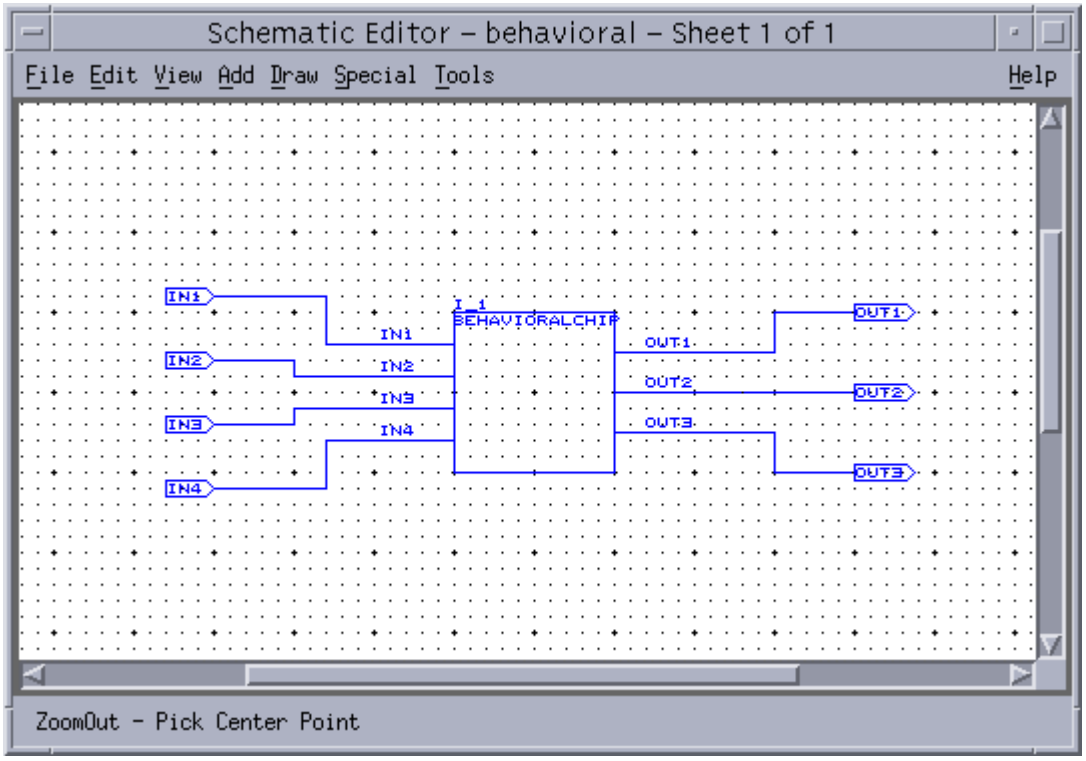


Figure 3-3: A Behavioral Schematic Drawn in SGE

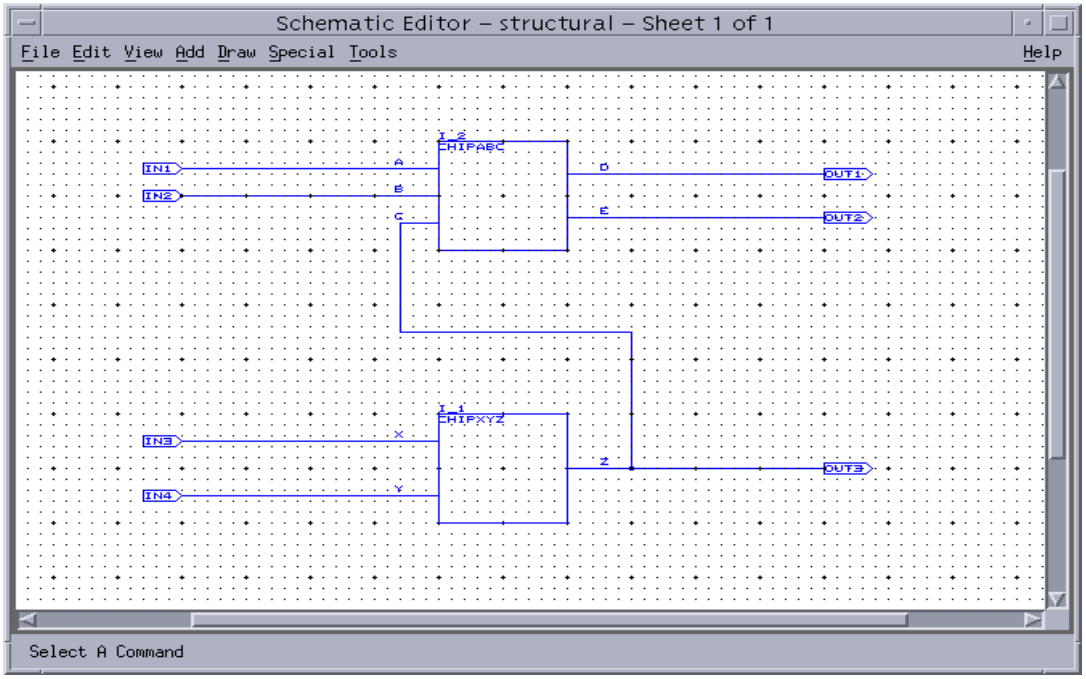


Figure 3-4: A Structural Schematic Drawn in SGE

```
-- VHDL Model Created from SGE Symbol behavioralchip.sym
-- Apr 18 14:21:25 1998

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_components.all;

entity BEHAVIORALCHIP is
   Generic ( DELAY:TIME:=3 NS );
      Port (     IN1 : In    BIT := 0;
                 IN2 : In    BIT := 0;
                 IN3 : In    BIT := 0;
                 IN4 : In    BIT := 0;
                OUT1 : Out   BIT := 0;
                OUT2 : Out   BIT := 0;
                OUT3 : Out   BIT := 0 );
end BEHAVIORALCHIP;

architecture BEHAVIORAL of BEHAVIORALCHIP is
   begin
end BEHAVIORAL;

configuration CFG_BEHAVIORALCHIP_BEHAVIORAL of BEHAVIORALCHIP is
   for BEHAVIORAL
   end for;
end CFG_BEHAVIORALCHIP_BEHAVIORAL;
```

Figure 3-5: The Initial Behavioral VHDL Model File

```
-- VHDL Model Created from SGE Schematic structural.sch
-- Apr 18 13:41:57 1998
library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_components.all;
entity STRUCTURAL is
      Port (     IN1 : In    BIT;
                 IN2 : In    BIT;
                 IN3 : In    BIT;
                 IN4 : In    BIT;
                OUT1 : Out   BIT;
                OUT2 : Out   BIT;
                OUT3 : Out   BIT );
end STRUCTURAL;
architecture SCHEMATIC of STRUCTURAL is
   signal     OUT3_DUMMY : BIT;
   component CHIPXYZ
   Generic ( DELAY:TIME:=3 NS );
      Port (       X : In    BIT := 0;
                   Y : In    BIT := 0;
                   Z : Out   BIT := 0 );
   end component;
   component CHIPABC
   Generic ( DELAY:TIME:=0 NS );
      Port (       A : In    BIT := 0;
                   B : In    BIT := 0;
                   C : In    BIT := 0;
                   D : Out   BIT := 0;
                   E : Out   BIT := 0 );
   end component;
begin
   OUT3 <= OUT3_DUMMY;
   I_1 : CHIPXYZ
   Generic Map ( DELAY=>3 NS )
      Port Map ( X=>IN3, Y=>IN4, Z=>OUT3_DUMMY );
   I_2 : CHIPABC
   Generic Map ( DELAY=>0 NS )
      Port Map ( A=>IN1, B=>IN2, C=>OUT3_DUMMY, D=>OUT1, E=>OUT2
);
end SCHEMATIC;
configuration CFG_STRUCTURAL_SCHEMATIC of STRUCTURAL is
   for SCHEMATIC
      for I_1: CHIPXYZ
         use configuration WORK.CFG_CHIPXYZ_BEHAVIORAL;
      end for;
      for I_2: CHIPABC
         use configuration WORK.CFG_CHIPABC_BEHAVIORAL;
      end for;
   end for;
end CFG_STRUCTURAL_SCHEMATIC;
```

Figure 3-6: The Initial Structural VHDL Model File

Once the modeler has both the input files, the specification document text file and the initial VHDL file, he/she can start the modeling process using the ModelMaker tool. It is suggested that both of these files be placed in the same directory with the ModelMaker executable file.

The modeler starts the ModelMaker tool by pressing the ModelMakerIII (which is linked to the ModelMakerIII.exe file) on his/her Windows Explorer window. This action executes the ModelMaker tool. At this point, the modeler must specify the location of the input files. He/she can do this by simply typing the file names on the space provided or finding the files using the "Browse" buttons on the dialog box shown in Figure 3-7. In this example, the specification document file is the "dma8237a.txt" file, and the basic VHDL file is the "dma8237a.vhd" file.

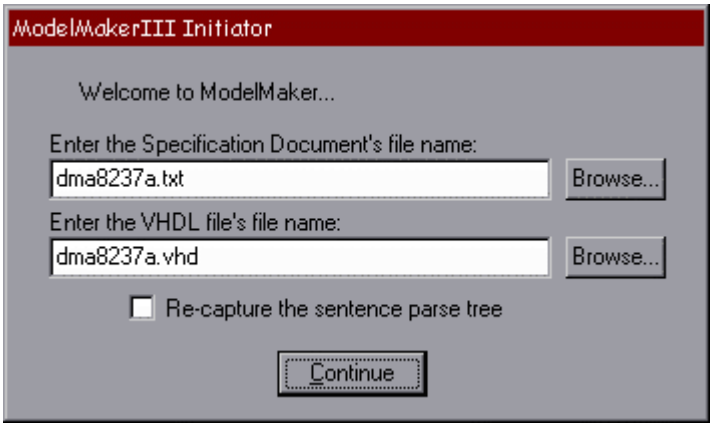


Figure 3-7: The Dialog Box Used to Specify the Location of the Files

Once the "Continue" button is pressed, ModelMaker starts its actual processing. Depending on the processing power of the machine used, this initial processing may require some time. During this initialization time, ModelMaker creates the specification document object, searches for keywords, and compiles the search result database for each of the keywords.

Internally, the first thing that ModelMaker performs during the initial processing is checking for the "ModelMakerIII.log" setup file. This file is created when

ModelMaker is executed for the first time. In the current implementation of ModelMaker, this configuration file is used only to store the path location to the SWI Prolog interpreter. For future versions, however, this configuration file can be used to store configuration information. If this file does not exist, a dialog box is displayed to ask the modeler for the location of the SWI Prolog interpreter. If the file exists, it will be opened and used to configure the ModelMaker tool.

Next, ModelMaker runs its internal initialization procedure. This procedure includes creating the Model Window, Source Window, and other controls, initializing all of the internal variables and pointers, and establishing the default setup.

The second activity that ModelMaker performs during the initial processing is creating the specification document object. Internally, this specification document object is called m_pSpec, and is an instance of the CSpecification class.

The constructor-member-function[3] of the CSpecification class processes the source specification document and the initial VHDL model files. First, the CSpecification constructor creates the grammar and dictionary objects that are needed for the processing. These objects are the aGrammar (of CGrammar class) object, the aDictionary (of CDictionary class) object, and the aMDictionary (of CMWDictionary class) object. More information on how these objects are created and loaded can be found in section 4.3.1.5, 4.3.1.3, and 4.3.1.4. The aGrammar object is created to store the grammar knowledge for the m_pSpec object. The aDictionary object and the aMDictionary object are created to store the standard dictionary and the multi-word dictionary knowledge for the m_pSpec object.

The input file that contains the grammatical knowledge is the “gram.txt” file. The CGrammar::Load() member function is called by the CSpecification constructor-member-function to initialize the CGrammar object. This CGrammar::Load() function

[3] Constructor member functions are special member functions that are called when class objects are created [24].

reads the input file and places the grammatical knowledge into its internal array of grammar rule objects. The Noun Phrase Extractor needs this array to perform its parsing process. Refer to section 4.3.1.5 for more information about this CGrammar class.

The aDictionary object is created to store the standard dictionary. In calling the CDictionary constructor, an argument specifies the file name of the dictionary file to be used. This argument is needed since more than one Dictionary object is used in ModelMaker. The standard dictionary file used to create aDictionary object is the "rdict.txt" file. To illustrate this dictionary object, some portion of the aDictionary object is presented in Figure 3-8. Section 4.3.1.3 gives more information about this CDictionary class.

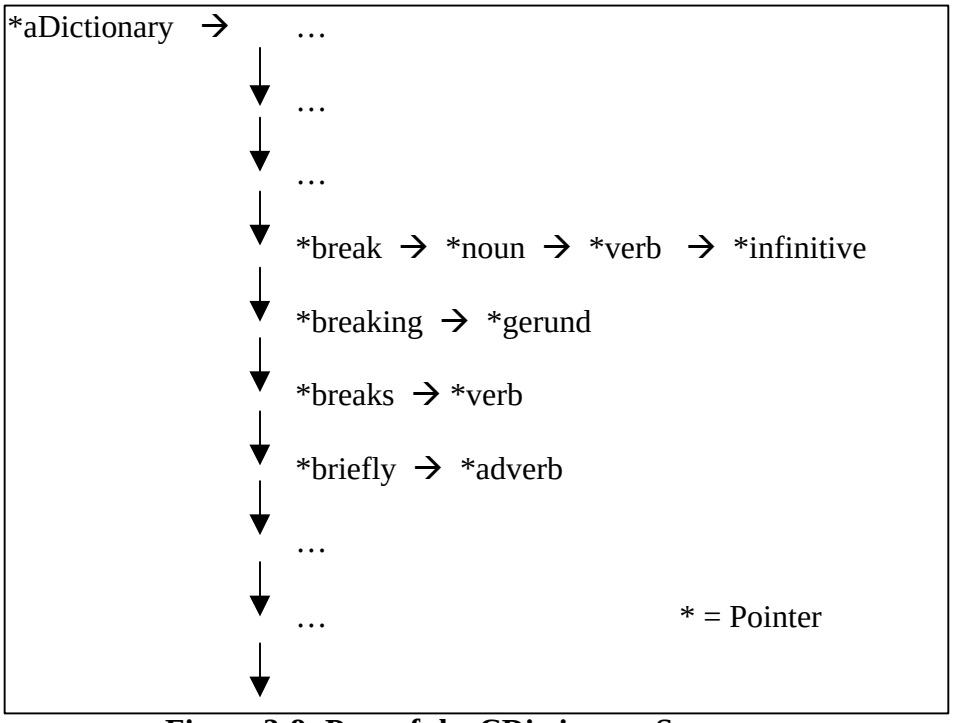


Figure 3-8: Part of the CDictionary Structure

Similarly, the CMWDictionary object is created by calling the CMWDictionary constructor-member-function. The "MW" in the name of this class stands for Multi Word. This object is created to store the multi-word dictionary knowledge. However, this CMWDictionary class is closer to the CGrammar class in its implementation,

because this class is a static class that does not need any outside information. There is no argument needed for the CMWDictionary constructor-member function. As in the case with the CGrammar class, the CSpecification class constructor-member-function calls the CMWDictionary::Load() to initialize the CMWDictionary object. The input file that contains all of the grammar knowledge is set to be the "mdict.txt" file. To illustrate this multi-word dictionary object, some portion of the aMDictionary object is presented in Figure 3-9. More information about this CGrammar class can be found in section 4.3.1.4.

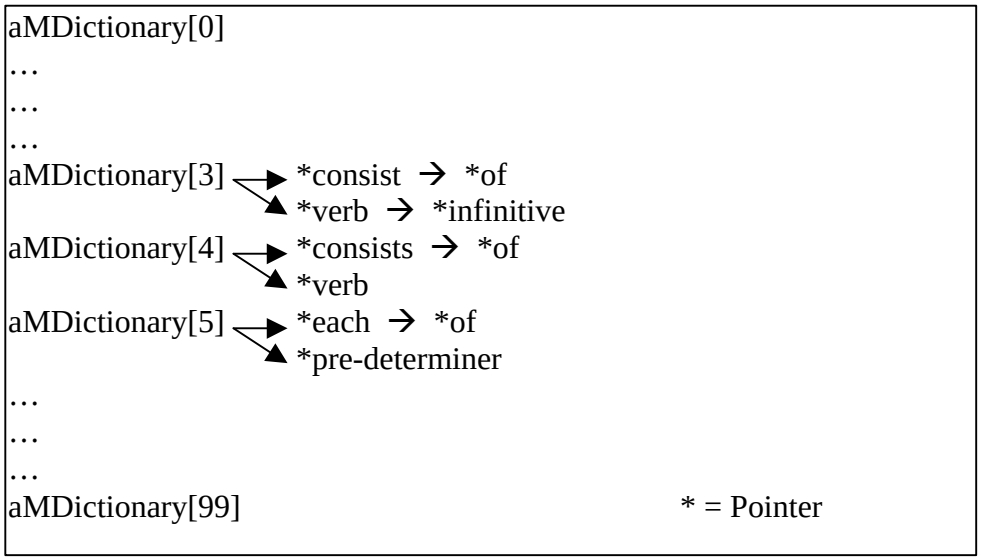


Figure 3-9: Part of the CMWDictionary Structure

As soon as these grammar and dictionary objects are created, an object of the CChart class, called aChart, is created. The aChart object is used to store the constituents of the sentences. The grammar and the dictionary objects need to be instantiated first, as the CChart class needs to use these objects as its knowledge database. Pointers to the aGrammar object, the aDictionary object, and the aMDictionary object are passed to the aChart object through the CChart constructor. The aChart object created by the constructor is an empty CChart object. The task of the constructor is only to prepare this CChart object. Nodes and constituents are added and deleted later by the m_pSpec object.

The Noun Phrase Extractor's Parser, which is used by the m_pSpec object, is an extension of the ASPIN bottom-up, parallel chart parser [23]. The parser algorithm is called a parallel algorithm, because it considers all possible alternatives at once. This Parser uses the standard and multi-word dictionary objects to create a chart from the document tokens (words, numbers, punctuation). The Parser searches in the dictionaries for the lexical category of each token and builds new constituents, as combination of the particular token and the existing constituents, based on the grammar rules. The grammar is an array of rules, which governs the combining process for constructing new constituents.

After aChart is constructed, two more dictionary objects of the CDictionary class are created for the m_pSpec object. These objects are the aBehaviorDic and the aStructuralDic. These dictionaries of behavioral and structural words are used to order the search results. The entries for these objects are loaded from the "behavior.dic" file and "structure.dic" files, respectively.

While performing its tasks, the m_pSpec object checks for any unsuccessful functions. If any functions returns an error code, the m_pSpec object will display an error message for the modeler and stop all of its processing. These errors include illegal dictionary format, dictionary file not found, and any other errors that deal with the input files. At this point, if no error is found, the m_pSpec object starts to load the initial VHDL model file and the source document file. This is done by calling the CSpecification::LoadVhd() and the CSpecification::LoadTxt() private member functions.

The task of the CSpecification::LoadVhd() function is to load the initial VHDL model file provided by the modeler into m_pSpec internal variable. While performing this task, this function also extracts all of the VHDL identifiers found in the VHDL file. These identifiers are then stored. This list is used by the m_pSpec object as the identifier keywords. For more information on the CSpecification::LoadVhd() private member function, see Main Member Function of CSpecification Class section (section 4.3.2.1).

The CSpecification::LoadTxt() function loads the source document file in the m_pSpec object. This task is not simple, for the function needs to identify the correct format of the specification document. This function is also responsible for calling the CChart::AddNode() member function, the CChart::ClearChart() member function, and the CChart::ParseNP() member function for the aChart object serving the m_pSpec object. Section 4.3.2.1 discusses the CSpecification::LoadTxt() private member function in more detail. That section also discusses the structure (sections, paragraphs, and sentences) of the source document object.

The CChart::AddNode() function is called for each token in the source document file. When this function is called from the aChart object, a new node is created inside the aChart object. This CChart::AddNode() function is also responsible for creating new constituents that combine the new node with the existing constituents of earlier nodes. This is done internally by the CChart class using the dictionary knowledge and the grammar knowledge built earlier for the CChart class. For more information about the CChart::AddNode(), refer to section 4.3.2.2.

The CChart::ClearChart() public member function is needed to reset the aChart object. Each time a new sentence is detected by the m_pSpec object, it clears the aChart object. Once this CChart::ClearChart() public member function is called, the aChart object is reinitialized to an empty chart. All of the existing node objects and constituent objects will be deleted, and all of the memory used will be reclaimed. This clean-up process is needed to limit the memory space consumed by the CChart object.

ModelMaker can also save the aChart in an ASCII file. The check box shown in Figure 3-7, with the caption “re-captured the sentence parse tree,” is provided to activate this capability. This capability is set to active only when the modeler clicks on this check box. When this capability is active, the entire contents of the aChart object are captured and appended into a string object called “m_Tree” just before the aChart object is cleared. This object is captured using a chart format, as shown in Figure 3-10. This capability is not active, by default, because the capturing process of the parse tree

requires a significant amount of time. In processing the 8237A DMA controller specification document having 5258 words, ModelMaker required approximately four hours to run on a 300 MHz Pentium processor. When the check box is not clicked, however, ModelMaker allows the modeler to view a preprocessed parse tree. To use this ability, the modeler needs to save the previously analyzed parse tree as the “tree.txt” file located in the same directory with the ModelMaker tool executable.

```
Node: #7   may
   Part: #35, Rule 0: mod, _, From: 7, To: 7
         may
Node: #10   be
   Part: #36, Rule 0: bei, _, From: 10, To: 10
         be
Node: #11   expanded
   Part: #37, Rule 0: ven, _, From: 11, To: 11
         expanded
   Part: #40, Rule 44: pvs, _, From: 7, To: 11, Parts: 35 36 37
         may be expanded
   Part: #41, Rule 165: nvs, _, From: 11, To: 11, Parts: 37
         expanded
   Part: #42, Rule 20: pred, _, From: 7, To: 11, Parts: 40
         may be expanded
   Part: #43, Rule 140: cl, _, From: 11, To: 11, Parts: 41
         expanded
   Part: #44, Rule 7: sp, _, From: 1, To: 11, Parts: 3 23 25 42
         it contains four independent channels and may be expanded
   Part: #45, Rule 70: n, _, From: 11, To: 11, Parts: 43
         expanded
   Part: #46, Rule 137: rest, _, From: 11, To: 11, Parts: 43
         expanded
```

Figure 3-10: Sample of the CChart Structure

The operation done by the CChart::ParseNP() function is performed just before the m_pSpec object clears the aChart object. This function extracts all of the noun phrases from the aChart object. These noun phrases are used by m_pSpec object as the noun phrase keywords. More information about the CChart::ParseNP() public member function can be seen in section 4.3.2.2.

Another operation done by the m_pSpec object before it clears the aChart object is to call the CChart::ParseSentences() public member function. CChart::ParseSentences() extracts all of the sentence parse trees found in the existing chart and writes them to the Template Constructor input file. This operation is essential

for the Template Constructor component, which analyzes the categories of sentences, based on the syntactic structure represented in the parse trees.

Once the CSpecification::LoadVhd() and the CSpecification::LoadTxt() private member functions are executed successfully, m_pSpec is ready to perform the pre-search to construct the search indexes. With the CSpecification::LoadVhd(), the m_pSpec object received the VHDL identifiers used in the initial VHDL model file. Moreover, with the CSpecification::LoadTxt(), the m_pSpec object received an array of strings that contain all of the noun phrases from the source specification document file. These noun phrases are also used by the m_pSpec object as the keywords for its pre-search.

From each of the words that came from these keywords, pointers to the element location where that particular word exists in the source document object are created by the m_pSpec object. A function of the CSpecification class parses the source document object and creates pointers to point to the elements that use the word from the keywords. These pointers allow the m_pSpec object to quickly find the existence of any keywords in the specification-document data-structure. These searches are done by the CSpecification constructor to allow quicker response to the modeler's queries during the modeling time.

This pre-search processing concludes the operation of the CSpecification constructor-member function. Once this CSpecification constructor-member function is executed successfully, the CSpecification object m_pSpec is ready for the ModelMaker tool. If no error occurs during the constructing process of this m_pSpec object, the ModelMaker tool will continue its initial processing.

The last thing that the ModelMaker tool performs during initial processing is updating the edit windows, the keywords and options combo boxes, and the info box. The source edit window is initialized with a predefined value (a clear screen that says "*** Specification Text Window ***"), and the model edit window is initialized with the basic VHDL file. The keywords combo box is initialized with the keywords list. Moreover, the search-type is set for Boolean "And" search and the sort-mode is set for

"Sequential" sort by default. Once this is done, the ModelMaker tool is ready to be used by the modeler. At this time, the window shown in Figure 1-2 is presented to the modeler.

From this point, there are many ways that the modeler can perform his/her modeling process. If a structural VHDL file is used, the modeler can start the modeling process by focusing on one component at a time. For each of these components, he/she could perform keyword searches that bring back information about that particular component. If a behavioral VHDL file is derived instead, the modeler can start by searching on the input and output identifiers of the chip. Each time a process or a function is suggested by the sentences returned, the modeler can model the process or function inside the behavioral model. Whether the modeler is using the structural VHDL file or the behavioral VHDL file, the modeler can always perform searches on any keyword. The keyword can be chosen from the keyword list, selected from the Model Window, or typed in the Keywords Window. Because the search results are displayed at the same level with the VHDL file, the modeler can do his/her modeling in the model window while referring to the information displayed in the source window.

Some methods used in the modeling process of the behavioral 8237A DMA controller model are presented below to illustrate the modeling process with the ModelMaker tool. The schematic and initial VHDL model file for this behavioral model is presented in Appendix B.

This is an illustration of the building process for a model of the data bus. In this case, the modeler is interested in modeling the data bus operation for the system. The modeler can start the modeling process by performing a search on the phrase "data bus." The modeler can either choose this phrase from the noun phrase keyword list or simply type the phrase in the Keywords Windows. Once this search is performed, results similar to the one shown in Figure 3-11 will be presented in the Source Window.

Within this search result, the modeler may notice at least three important sentences. These sentences are “During DMA cycles the most significant 8 bits of the address are output onto the data bus to be strobed into an external latch by ADSTB,” “In memory-to-memory operations, data from the memory comes into the 8237A on the data bus during the read-from-memory transfer,” and “When the processor replies with a HLDA signal, the 8237A takes control of the address bus, the data bus, and the control bus.”

```
<KEY:data:bus>

0 pin description

0.4 cs

this allows cpu communication on the data bus .

0.9 db0-db7
db0-db7 , in/out , data bus : the data bus lines are bidirectional three-state
signals connected to the system data bus .
during dma cycles the most significant 8 bits of the address are output onto the
data bus to be strobed into an external latch by adstb .
in memory-to-memory operations , data from the memory comes into the 8237a on the
data bus during the read-from-memory transfer .
in the write-to-memory transfer , the data bus outputs place the data into the new
memory location .

2 dma operation

2.1 idle cycle

the commands do not make use of the data bus .

2.2 active cycle

2.3 transfer types

2.3.5 address generation

3 register description

3.10 software command

they do not depend on any specific bit pattern on the data bus .

5 application information ( note 1 )
when the processor replies with a hlda signal , the 8237a takes control of the
address bus , the data bus and the control bus .
the address for the first transfer operation comes out in two bytes ; the least
significant 8 bits on the eight address outputs and the most significant 8 bits on
the data bus .
the contents of the data bus are then latched into an 8-bit latch to complete the
full 16 bits of the address bus .
```

Figure 3-11: ModelMaker Search Result on the Phrase “data bus”

The first sentence shows the data that needs to be sent to the data bus during DMA cycles. Similarly, the second sentence shows which signals is in the data bus during the read-from-memory transfer. The third sentence, the most important one, shows that the operation in the data bus can be done only after the processor sent out the HLDA signal. Based on this information, the modeler can start adding his/her process to the initial VHDL model. The modeler can call it the data_bus process and use the HLDA signal in its sensitivity list. This new process is illustrated in Figure 3-12.

```
data_bus_process: process (HLDA)

begin

        if DMA_CYCLE then
                DATA_BUS <= ADDRESS(HI_8_BITS);
        end if;

        if READ_FROM_MEMORY then
                DATA_BUS <= DATA(MEMORY);
        end if;

end process;
```

Figure 3-12: VHDL Process Created Based on Search Phrase "data bus"

The process shown in Figure 3-12 is certainly not a final VHDL process. The DMA_CYCLE and READ_FROM_MEMORY activities need to be defined as signals and the HI_8_BITS of MEMORY address need to be specified. This VHDL process, however, can serve as a starting point where the modeler can build his/her model. The modeler can continue his/her modeling by performing searches on the signals that he/she needs in this initial process.

The second illustration, building a process based on information about a pin. In this case, the modeler is interested in modeling the system from the external pins. The modeler can pick a pin and try to model the necessary process for the particular pin. In

this illustration, the modeler starts from the "/EOP" pin. Again, the modeler can either choose this phrase from the keyword list or simply type the phrase in the Keywords Windows. Once this search is performed, a result similar to the one shown in Figure 3-13 will be presented in the Source Window.

```
<KEY:/eop>

each channel can be individually programmed to autoinitialize to its original
condition following an end of process ( /eop ) .

0 pin description

0.12 /eop
/eop , in/out , end of process : end of process is an active low bidirectional
signal .
information concerning the completion of dma services is available at the
bidirectional /eop pin .
this is accomplished by pulling the /eop input low with an external /eop signal .
this generates an /eop signal which is output through the /eop line .
the reception of /eop , either internal or external , will cause the 8237a to
terminate the service , reset the request , and , if autoinitialize is enabled , to
write the base registers to the current registers of that channel .
the mask bit and tc bit in the status word will be set for the currently active
channel by /eop unless the channel is programmed for autoinitialize .
during memory-to-memory transfers , /eop will be output when the tc for channel 1
occurs .
/eop should be tied high with a pull-up resistor if it is not used to prevent
erroneous end of process inputs .

2 dma operation
2.2 active cycle
2.2.2 block transfer mode
in block transfer mode the device is activated by dreq to continue making transfers
during the service until a tc , caused by word count going to ffffh , or an
external end of process ( /eop ) is encountered .

2.2.3 demand transfer mode
in demand transfer mode the device is programmed to continue making transfers until
a tc or external /eop is encountered or until dreq goes inactive .
only an /eop can cause an autoinitialize at the end of the service .
/eop is generated either by tc or by an external signal .

2.3 transfer types
2.3.1 memory-to-memory
when the word count of channel 1 goes to ffffh , a tc is generated causing an /eop
output terminating the service .
the 8237a will respond to external /eop signals during memory-to-memory transfers .

2.3.2 autoinitialize
during autoinitialize initialization the original values of the current address and
current word count registers are automatically restored from the base address and
base word count registers of that channel following /eop .
if interrupted externally , /eop pulses should be applied in both bus cycles .
.
.
.
```

Figure 3-13: ModelMaker Search Result on the Word "/eop"

From the search result shown in Figure 3-13, the modeler can observe significant actions that are connected to the word “autoinitialize.” To learn more about “autoinitialize,” the modeler can then use this word as the next search word. Using this new search word, a result similar to the one shown in Figure 3-14 will be shown in the Source Window.

```
<KEY:autoinitialize>

each channel can be individually programmed to autoinitialize to its original
condition following an end of process ( /eop ) .

0 pin description
0.12 /eop
the reception of /eop , either internal or external , will cause the 8237a to
terminate the service , reset the request , and , if autoinitialize is enabled , to
write the base registers to the current registers of that channel .
the mask bit and tc bit in the status word will be set for the currently active
channel by /eop unless the channel is programmed for autoinitialize .

2 dma operation
2.2 active cycle
2.2.1 single transfer mode
when the word count rolls over from zero to ffffh , a terminal count ( tc ) will
cause an autoinitialize if the channel has been programmed to do so .
2.2.3 demand transfer mode
only an /eop can cause an autoinitialize at the end of the service .
2.3 transfer types
2.3.2 autoinitialize
by programming a bit in the mode register , a channel may be set up as an
autoinitialize channel .
during autoinitialize initialization the original values of the current address and
current word count registers are automatically restored from the base address and
base word count registers of that channel following /eop .
the mask bit is not altered when the channel is in autoinitialize .
following autoinitialize the channel is ready to perform another dma service without
cpu intervention , as soon as a valid dreq is detected .
in order to autoinitialize both channels in a memory-to-memory transfer , both word
counts should be programmed identically .

3 register description
3.1 current address register
it may also be reinitialized by an autoinitialize back to its original value .
autoinitialize takes place only after an /eop .
3.3 base address and base word count registers
during autoinitialize these values are used to restore the current registers to
their original values .
3.7 mask register
each mask bit is set when its associated channel produces an /eop if the channel is
not programmed for autoinitialize .
```

Figure 3-14: ModelMaker Search Result on the Word “autoinitialize”

From the “autoinitialize” search result, the modeler may notice at least one crucial item. The sentence that carries that information says “During autoinitialize initialization the original values of the current address and current word count registers are automatically restored from the base address and base word count registers of that channel following /EOP.” Based on this information, the modeler can build his/her autoinitialize process. The sentence specifies the signal that triggers the actions and the actions that need to be done. With this knowledge, the modeler can build a VHDL process as shown in Figure 3-15.

```
autoinitialize_process: process (EOP)

begin

    if EOP'event and EOP='0' then
       CURRENT_ADDRESS <= BASE_ADDRESS;
       CURRENT_WORD_COUNT <= BASE_WORD_COUNT;
    end if;

end process;
```

Figure 3-15: VHDL Process Created Based on Search Phrase “autoinitialize”

The two examples presented above illustrate the capability of the ModelMaker tool with the Noun Phrase Extractor. To show the usability of the Template Constructor, an illustration is presented below.

Suppose the modeler is trying to verify his understanding of the sentences from the source specification document. By pressing the “Template-View” button, the Template Constructor will return all of the analyzable sentences in the Source Window. If the sentence in question is an analyzable sentence, the modeler can use the information from the Template Constructor to decipher the sentence. For example, Figure 3-16 shows one of the analyzed sentences from the Template Constructor.

=== Sentence #330
the device will output an hrq to the microprocessor and enter the active cycle .

1 AND < *behavior*
  and: **ENTER** (2)
  and: **OUTPUT** (3)
2 ENTER < *call* < *event*
  enters: **CYCLE** (4)
  agent: **DEVICE**, the device
4 CYCLE < *state* < *behavior*
  attribute: **ACTIVE**
3 OUTPUT < *move* < *action*
  destination: **MICROPROCESSOR** < *processor* < *device*, the microprocessor
  operand: **HRQ** < *id*, an hrq
  agent: **DEVICE**, the device
  modal: **WILL**

Figure 3-16: Sample Result from the Template Constructor

The Template Constructor result shown in Figure 3-16 came from the sentence "The device will output an HRQ to the microprocessor and enter the active cycle." The resulting template shows that the *device* will perform two operations. The operations are *enter* and *output*. The whole template can be interpreted in two sentences, "the device will enter the active cycle" and "the device will send an HRQ signal to the microprocessor." The modeler can then use these sentences to verify his/her model.

CHAPTER 4: MODELMAKER BASIC DESIGN

4.1 Introduction

ModelMaker integrates a search engine and two components from a natural language understanding system with a graphical user interface providing an editor and search controls. The natural language processing components are the Parser and the Semantic Analyzer. These components are taken from the ASPIN system [12]. They were enhanced and customized from the original versions to serve in ModelMaker. The new versions are called the Noun Phrase Extractor and the Template Constructor, respectively. ModelMaker provides a common environment for both of these tools, and adds more functionality to build a new tool for rapid modeling of a device from informal specifications. This new functionality includes the Document Reader, the VHDL Identifier Extractor, and a search ability based on the knowledge it builds during initialization and run time.

In object-oriented terms, ModelMaker can be separated into many objects. The graphical user interfaces are objects from the Microsoft CDialog class. While ModelMaker is running, at least three of these CDialog objects are created. The first one is used to query the modeler about the input files. This object is created right after the modeler clicks on the ModelMaker icon. Once the modeler specifies the input files, this CDialog object is destroyed, and an object of the CSpecification class is created. As the creation process of this CSpecification class is silent (there is no visual indicator), a second object from the CDialog class is created to inform the modeler that processing will take some time. This CDialog object is destroyed as soon as the CSpecification object is created successfully. The third CDialog object, the main ModelMaker window, is then created right after the second CDialog object is deleted.

The CSpecification object is the main object of the ModelMaker tool. This object creates all of the other objects and brings them together to create a useful tool. The creation process of this CSpecification object may require some time, depending on the

size of the input files and the processing power of the machine used. This is because the CSpecification object is responsible not only for creating the other necessary objects but also for loading the input files and analyzing them for the needed information. A CLexer object is created by this CSpecification object to divide the input text file into tokens (words, punctuation, and numbers). Three CDictionary objects, one CMWDictionary object, and one CGrammar object are created as knowledge databases for the CSpecification object. Moreover, a CChart object is created to handle the noun phrase extracting task. Finally, the source specification document is stored in a hierarchy of CElement objects representing sentences, paragraphs and sections of the source document.

The Semantic Analyzer component is not implemented as an object. This is because the Semantic Analyzer component is implemented in Prolog. When the Semantic Analyzer is needed by the ModelMaker tool, a system call to the Prolog interpreter is executed. The communication between the CSpecification object and the Semantic Analyzer is done by means of input and output files.

In this chapter, a detailed object-oriented description of ModelMaker is presented. To give more insight about the system, the main ModelMaker files will be listed and explained first. Following the explanation of the files, the main classes, and member functions, and the intercommunications among them will be discussed in a more object-oriented way.

Because the Noun Phrase Extractor and the Template Constructor were based on existing research work prior to this research, each of these components will be explained with more detail in a separate chapter.

4.2 File Descriptions

4.2.1 C++ Header and Source Files

ModelMaker source files can be separated into two different parts. They are the graphical user interface (GUI) part and the engine part. The GUI part deals with the user interaction such as capturing user's inputs and displaying results back to the user. This first part is written with the Microsoft Visual C++ [18]. The Microsoft Foundation Class (MFC) library version 4.21 is used to implement this interface. The second part, the engine, contains the actual code that is needed to perform the document analysis and searches. This includes extracting noun phrases from the specification text, extracting VHDL identifiers from the VHDL file, and searching the specification text for keywords. This engine is written under the ANSI C++ standard[4] for portability. This part can be re-compiled for a different platform without changing much of its code. The GUI will need a major re-build to port ModelMaker to a different platform. Some minor changes in the ModelMaker engine would be required since interfacing with the GUI required some Microsoft specific code.

[4] The final standard is not available for public access as of April 1998. The final draft of this standard, however, can be obtained from the American National Standards Institute (ANSI) Standards Secretariat: ITIC, 1250 Eye Street NW, Suite 200, Washington DC 20005. An Internet version of the final draft can be found in this URL: http://www.maths.warwick.ac.uk/cpp/pub/wp/html/cd2/. The URL of this document can be changed at any time by the web-master.

The necessary C++ header and source files to rebuild ModelMaker are listed in Table 4-1 and are explained in alphabetical order below.

Table 4-1: C++ Header and Source Files

Program Files	Header Files
Chart.cpp	Chart.h
Dictionary.cpp	Dictionary.h
Grammar.cpp	Grammar.h
Initiator.cpp *****	Initiator.h *****
Lexer.cpp	Lexer.h
MMREditWin.cpp *****	MMREditWin.h *****
ModelMakerIII.cpp *****	ModelMakerIII.h *****
ModelMakerIIIDlg.cpp *****	ModelMakerIIIDlg.h *****
MultiWord.cpp	MultiWord.h
PrologFinder.cpp *****	PrologFinder.h *****
Specification.cpp	Specification.h
StatusBox.cpp *****	StatusBox.h *****
StdAfx.cpp *****	StdAfx.h *****
	Resource.h *****
• ***** = Microsoft specific code (platform dependent code) • All others = ANSI C++ code	

Chart.cpp & Chart.h: These files contain classes needed to build the parse tree of each parsed sentence. Some of the important classes included in these files are CChart, CPart, and CNode. The CChart class is the main class in this file. It uses the other classes to store its information. The information from the CChart class is used by ModelMaker to extract the noun phrase segment of the sentence that will be used to build the noun phrase keyword list. The classes included in these files are the enhanced version of the classes used in the ASPIN sentence parser, the Parser. In ModelMaker, they are called the Noun Phrase Extractor.

Dictionary.cpp & Dictionary.h: These files contain classes needed to hold the dictionary structure used in ModelMaker. The main class in these files is the CDictionary class. Other important classes that are used by this class are the CEntry class and the CMeaning class. A CDictionary object can have an unlimited number of

entries (pointers to CEntry objects), which, in turn, can have an unlimited number of categories (pointers to CMeaning objects). Objects that are created from this CDictionary class include the main dictionary (aDictionary), the behavioral word dictionary (aBehaviorDic), and the structural word dictionary (aStructuralDic), which are all used in the CSpecification class (see section 4.3.1.1). The classes included in these files are also the enhanced version of the classes used in the ASPIN sentence parser.

Grammar.cpp & Grammar.h: The CGrammar class is the main class of these files. This class is used by ModelMaker to store the grammar rules that are used by the Noun Phrase Extractor. These grammar rules are stored as CRule objects. These classes are built upon the ASPIN sentence parser's classes.

Initiator.cpp & Initiator.h: These files house the CInitiator class. This class is inherited from the CDialog class of the Microsoft Foundation Class. An object of this class is used by ModelMaker to prompt the user for the input files' names. This class is platform dependent. It will work only under the Microsoft Visual C++ environment. For this reason, this class will not be explained in details.

Lexer.cpp & Lexer.h: The only class in these files is the CLexer class. This class is responsible for loading the input files, and dividing their character strings into tokens. A more descriptive definition of a token is presented in the section about the CLexer class (4.3.1.7). This class is built to support the specification document reader function of ModelMaker. An object of this class is utilized by the CSpecification method to read the specification document. By using this class, ModelMaker is able to distinguish between a period that acts as a sentence terminator and a period that acts as part of a decimal number.

MMREditWin.cpp & MMREditWin.h: These files house the CMMREditWin class. This class is inherited from the CRichEditCtrl class of the Microsoft Foundation Class. Similar to the CInitiator class, this class is platform dependent. It will work only under the Microsoft Visual C++ environment. ModelMaker uses two objects that are of

this class. The first one is the source edit window, and the second is the model edit window. This edit window is customized, as it has to be able to receive text with custom formatting tags created by the Template Constructor. Using this class for the edit window, ModelMaker is able to display text in italic, bold, and color.

ModelMakerIII.cpp & ModelMakerIII.h: These are the main files for the ModelMaker application in the Windows platform. These files contain the CModelMakerIIIApp class that is inherited directly from the CWinApp class of the Microsoft Foundation Class. The only function of this class is to start the application under the Windows environment.

ModelMakerIIIDlg.cpp & ModelMakerIIIDlg.h: The only class in these files is the CModelMakerIIIDlg class. When the ModelMaker application is launched, an object of this class is initialized by the CModelMakerIIIApp class. Once initialized, this object becomes the main ModelMaker user interface. All of the user interface control objects are governed by this CModelMakerIIIDlg object.

MultiWord.cpp & MultiWord.h: These two files contain the CMWDictionary class, the CMWEntry class, and the CMWMeaning class. All of these classes are ANSI C++ classes. The CMWDictionary is the main class in these files, and uses the other two classes as its elements. This CMWDictionary class is responsible for holding the multi-word dictionary structure. This is to handle phrases such as “consist of,” “faster than,” “greater than or equal to,” or any other multi-word phrases. The Noun Phrase Extractor will invoke this CMWDictionary object if it finds an asterisk symbol, “*”, listed as a word’s category from the standard CDictionary object. This CMWDictionary object will then compare any available phrase that ends with that word to the current phrase that the Noun Phrase Extractor is processing. If any match is found, this CMWDictionary object will return the category of the phrase back to the Noun Phrase Extractor. To use this capability, each of the last words in the multi-word phrases listed in the multi-word dictionary file (mdict.txt) needs to be listed in the standard dictionary file (rdict.txt) with a “*” in its category.

PrologFinder.cpp & PrologFinder.h: These files house the CPrologFinder class. Similar to the CInitiator class, this class is inherited from the CDialog class of the Microsoft Foundation Class. An object of this class is used by ModelMaker to prompt the user for the location of the SWI Prolog executable file. Once the SWI Prolog executable is found, ModelMaker generates a log file in the local directory that contains the location information for that file. For the next ModelMaker invocation, it will use the information contained in this file instead of creating another object from this CPrologFinder class.

Specification.cpp & Specification.h: These files house the main ANSI C++ classes for the specification document reader, CSpecification. This CSpecification class is responsible for loading the specification document with a correct document hierarchy, extracting the keywords out of the specification document, and performing any other operations that deal with the specification document data structure. The other important class in these files is the CElement class. The CSpecification class uses a CElement class for each of its structural elements. The possible CSpecification elements are section, paragraph, and sentence. More details on these elements and the CSpecification class are presented in the CSpecification Class section (4.3.1.1).

StatusBox.cpp & StatusBox.h: These files contain the CStatusBox class. This class is inherited from the CDialog class of the Microsoft Foundation Class. An object of this class is invoked by ModelMaker only to give feedback to the user about its processing status.

StdAfx.cpp & StdAfx.h: These files are Microsoft Visual C++ specific files. They are used to connect the application to the standard Microsoft Foundation Class library.

Resource.h: This file is a Microsoft Visual C++ specific file. It is used to define all of the “include” lines for the application and to specify all of the other symbol definitions needed for the application.

4.2.2 Prolog Files

The initial version of the ASPIN Semantic Analyzer (SemAnal) used in the ModelMaker application was written in Quintus Prolog. Because converting the Semantic Analyzer from Quintus Prolog to the ANSI C++ environment was far beyond the objective of this research, it was decided to modify this Semantic Analyzer to run on a PC with SWI Prolog. This section presents some description of the Prolog files, which are listed in Table 4-2.

Table 4-2: Prolog Files

Files	Information Contained / Functions
Canon.pl	Meanings of words
Semrules.pl	Semantic Analysis rules
Roots.pl	Mapping tokens to root form
Main.pl	Semantic Analyzer engine

Canon.pl: This file contains the conceptual type hierarchy for dictionary words as well as the conceptual graph meanings for words that are currently covered by the vocabulary of the Semantic Analyzer. It also contains definitions for the routines to find the minimal semantic super-type of a concept.

Roots.pl: This file contains information about the root form of each word listed in the dictionary. The Semantic Analyzer uses this information to select semantics from the canon for a given word.

Semrules.pl: This file contains the semantic analysis rules used by the Semantic Analyzer to combine the meanings of words into sentence meanings. Each grammatical rule of the parser has a semantic analysis rule associated with it.

Main.pl: This file is the main file of the Semantic Analyzer. Once invoked, this file calls the rest of the Semantic Analyzer files. This file contains the procedures invoked by the semantic rules to combine word semantics.

4.2.3 Database Files

ModelMaker uses three database files to support parsing by the Noun Phrase Extractor. These files are the standard dictionary file (rdict.txt), the multi-word dictionary file (mdict.txt), and the grammar file (gram.txt). These files are stored as ASCII, space delimited files for portability reasons.

4.2.3.1 Standard Dictionary File

The standard dictionary file, "rdict.txt," contains a list of words along with each word's category. An annotated screen capture from a small part of this dictionary can be seen in Figure 4-1.

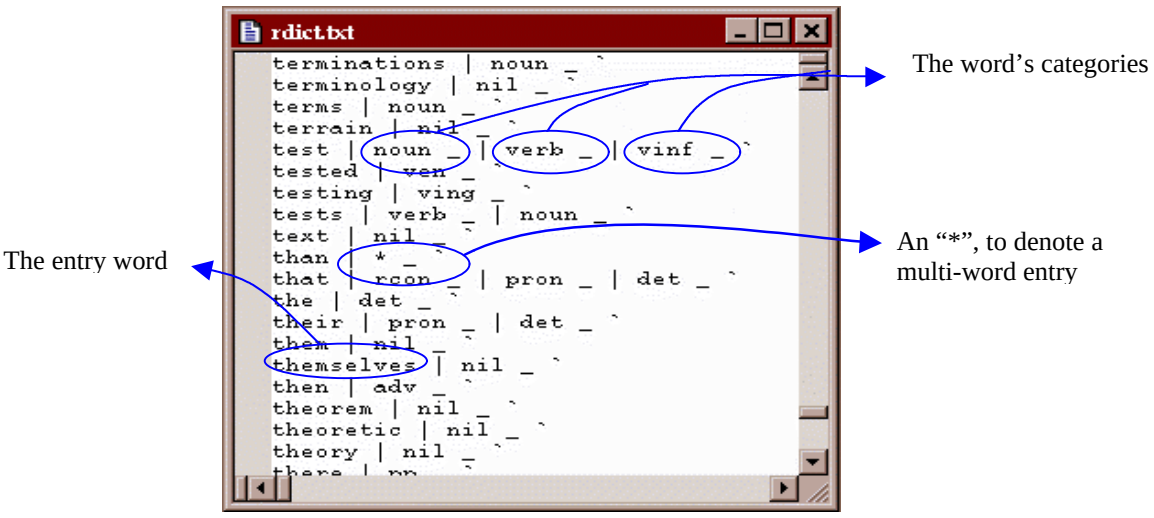


Figure 4-1: Captured and Annotated Standard Dictionary File

Each line in this file represents one entry in the dictionary. Each entry consists of a vocabulary word and a list of one or more lexical categories (parts-of-speech). Each category consists of a pipe "|" followed by a category indicator followed by an underscore. A grave accent mark, "`," marks the end of an entry.

An asterisk, in the category field denotes that one or more multi-word phrases that ended with the entry word are listed in the multi-word dictionary. Once the Noun Phrase Extractor finds this asterisk, it invokes the multi-word dictionary object to check for another possible category. For example, in Figure 4-1, the asterisk sign for the entry *than* refers to the multi-words *farther than, faster than, fewer than, greater than,* and *less than*, which are listed in the multi-word dictionary.

4.2.3.2 Multi-Word Dictionary File

The entry-field structure of the multi-word dictionary file, "mdict.txt," is similar to the entry-field structure of the standard dictionary file. Each line of this multi-word dictionary file also represents one multi-word entry in the dictionary. Each of the entry lines is also divided into two or more fields as in the standard dictionary file. The only difference between the two entry field structures is that the multi-word dictionary entry field structure allows more than one word in the first field. A screen capture from a part of this dictionary can be seen in Figure 4-2.

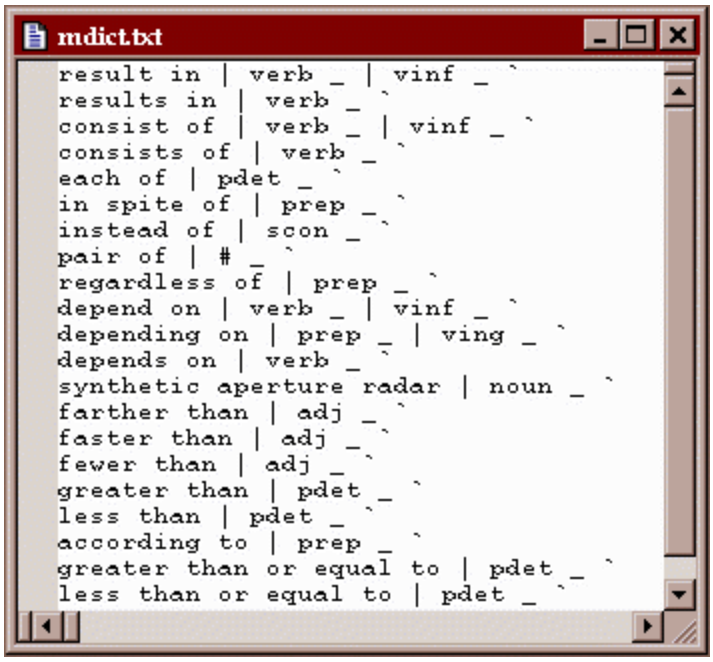


Figure 4-2: Captured Multi-Word Dictionary File

4.2.3.3 Grammar File

The grammar file, "gram.txt," contains the grammar rules needed by the Noun Phrase Extractor. This grammar rule file was originally designed for the ASPIN sentence parser, the Parser. A captured image of part of the grammar file is presented in Figure 4-3.

This grammar file contains five main fields on each line. Each of the fields is separated by a space character. The first field is called the rule ID field. This field contains the grammar rule identifier for this entry. The second field is the constituent count field. This field specifies the number of constituents that are needed to create the product of this rule. The constituents are listed in the fifth and subsequent fields. They are placed last to allow a variable number of constituents. The current Noun Phrase Extractor limits this variable to the range of one to six constituents. The third field is not used by ModelMaker. The fourth field is the result field. This field stores the new category of the combined constituents.

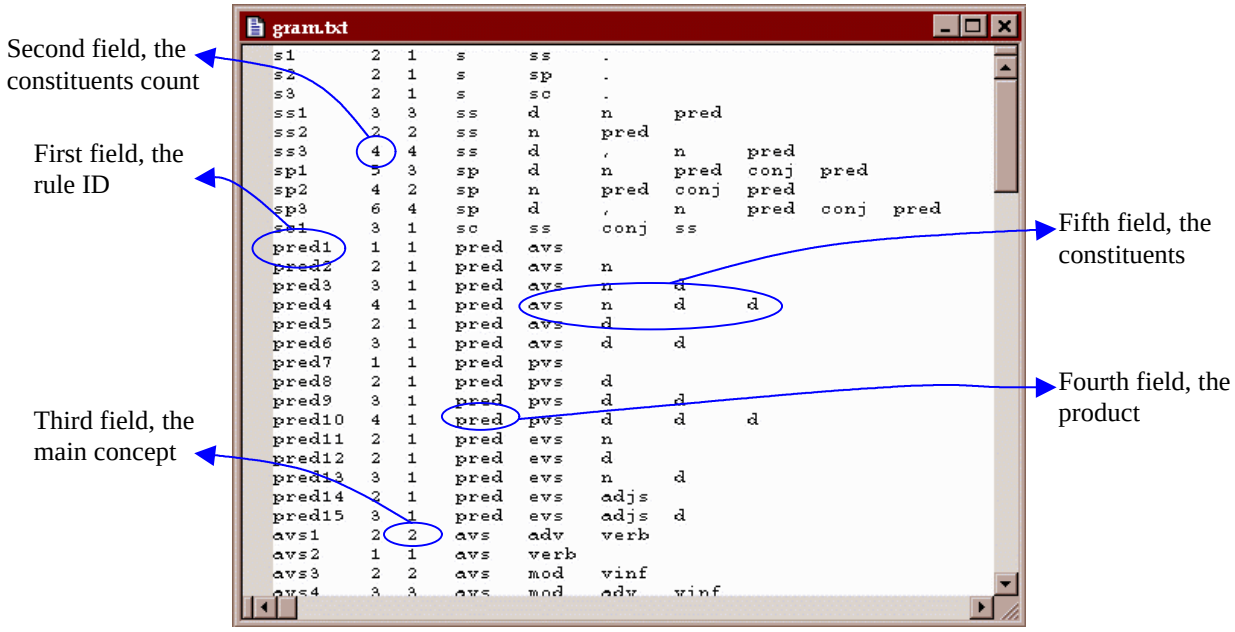


Figure 4-3: Captured & Annotated Grammar File

4.2.4 Input Files

ModelMaker requires two input files. These input files are the VHDL model file and the source text file.

4.2.4.1 VHDL Model File

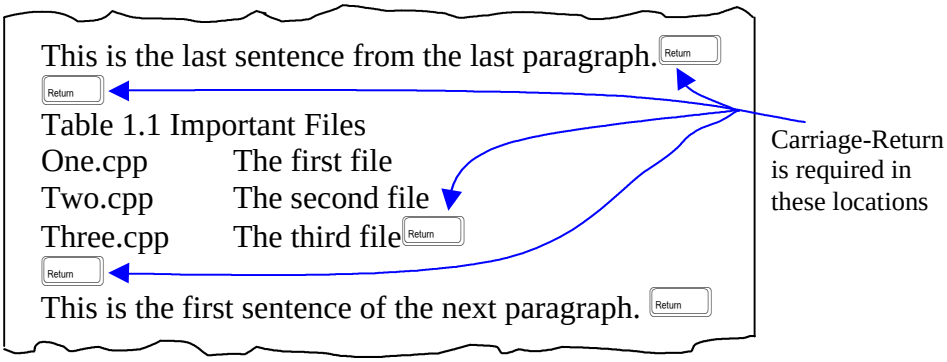
This file stores the VHDL code for the ModelMaker. The initial VHDL code can be written manually or extracted automatically from a schematic capture program such as Synopsys Graphical Environment (SGE) Shell from Synopsys. On initialization, ModelMaker loads this file, stores it as a local variable in the memory, and displays it to the user in the Model Window. During the loading process, ModelMaker also parses this file to extract all of the VHDL identifiers to be used as search keywords. More details on this Identifier Extractor can be found in section 4.3.2.1, Main Member Function of CSpecification Class.

4.2.4.2 Source Text File

This file ingests the ASCII specification document text for the device that the modeler wishes to model. In order to store this information in a document data structure, ModelMaker attempts to recognize the format of this specification document. At this time, the ModelMaker document reader can recognize only a limited variation of formats. ModelMaker separates the document into five element types: *section header*, *paragraph*, *sentence*, *table*, and *figure*. More detailed definitions of these element types are presented in the CSpecification Class section (4.3.1.1).

In dividing the document into those five elements, ModelMaker uses the following rules:

- Elements are separated by two line-feeds or two carriage-returns.
- Each section must have a section header (this rule can be violated for the first section).
- The section header must begin on a new line and be written in the format of "#[.#] *Caption.*" In this format, "#" represents one or more digits from zero to nine, and "Caption" can represent any string followed by two carriage-returns. The part inside the brackets can be omitted or repeated. Examples of valid section headers would include "1 Introduction", "2.1 Search Algorithm", or "2.1.3 Multiple Search".
- Tables and figures must be introduced by writing the word *Table* or *Figure* (whichever is more appropriate) as the first word. An example of a valid table element can be seen below.



- Any other element (group of sentences starting and ending with two carriage-returns) that is not recognized as a section, table or figure will be called a *paragraph*.
- A period is used to declare the end of a sentence. Each group of words that is inside a paragraph and ends with a period is called a sentence. An exception to this rule can occur with the last sentence in the paragraph. A paragraph can contain an unlimited number of sentences.

4.2.5 Output Files

ModelMaker does not create any permanent output files. However, it generates several temporary files. The main temporary file is the file called "suite.pl." This file is written automatically by ModelMaker during initialization. This file contains the parse trees of sentences in a Prolog format that will be used by the Template Constructor as input. This file is deleted automatically once ModelMaker is terminated. ModelMaker also generates files called "_out.pl," "_done.pl," and "_runsemanal.pl." The last two files are deleted automatically by ModelMaker. If ModelMaker is terminated abnormally and some of these two temporary files were not deleted, it is recommended that these files be deleted manually. The "_out.pl" file, however, is the output file generated by the Template Constructor. If the modeler is planning to use the "Template-View" capability in ModelMaker, he/she needs to keep this file in this ModelMaker directory. This file is generated when the modeler presses the "Template-Re-Analyze" button.

Though ModelMaker does not create any permanent output files, it provides the user with the ability to save the VHDL code back to the VHDL input file.

4.3 Object Oriented Description

ModelMaker is implemented in C++. This is done to ensure maintainable and expandable code, as ModelMaker is still in its prototype age. The concepts of inheritance, encapsulation, member functions and member variables (refer to the

Mastering Microsoft Visual C++ 4 [25], the On to C++ [24], and the Inside the C++ Object Model [17]) were used to achieve this goal. Using the object oriented paradigm, ModelMaker is presented with more details in this section. The main classes in ModelMaker are described in this section. This discussion is followed by descriptions of the main member functions of ModelMaker, and finally, a discussion about the interactions within instances of these main classes.

4.3.1 Main Class Descriptions

The primary object used by ModelMaker engine is an object (m_pSpec) of the CSpecification class. The CSpecification class is the class that uses and combines the other main classes such as CChart, CDictionary, CGrammar, CLexer, and CMWDictionary. This CSpecification class and those other main classes are presented in this section. Booch class diagrams [4] are used to illustrate this section. In this notation, a dotted irregular outline denotes a class. A line from class C1 to C2 with a dot on the C1 end and a number n indicates that C1 uses n objects from class C2.

4.3.1.1 CSpecification Class

As mentioned earlier, the CSpecification class is the main class in the ModelMaker. As shown in Figure 4-4, this class uses multiple objects of the other classes. On its instantiation, a CSpecification object creates the CDictionary objects, the CMWDictionary object, and the CGrammar object. The CDictionary objects are used to keep the information of the standard dictionary, the structural dictionary, and the behavioral dictionary. The CGrammar and CMWDictionary objects are used to keep the information of the standard grammar and the multi-word dictionary. The pointer to these CDictionary, CMWDictionary, and CGrammar objects are then passed to the CChart constructor to create the CChart object.

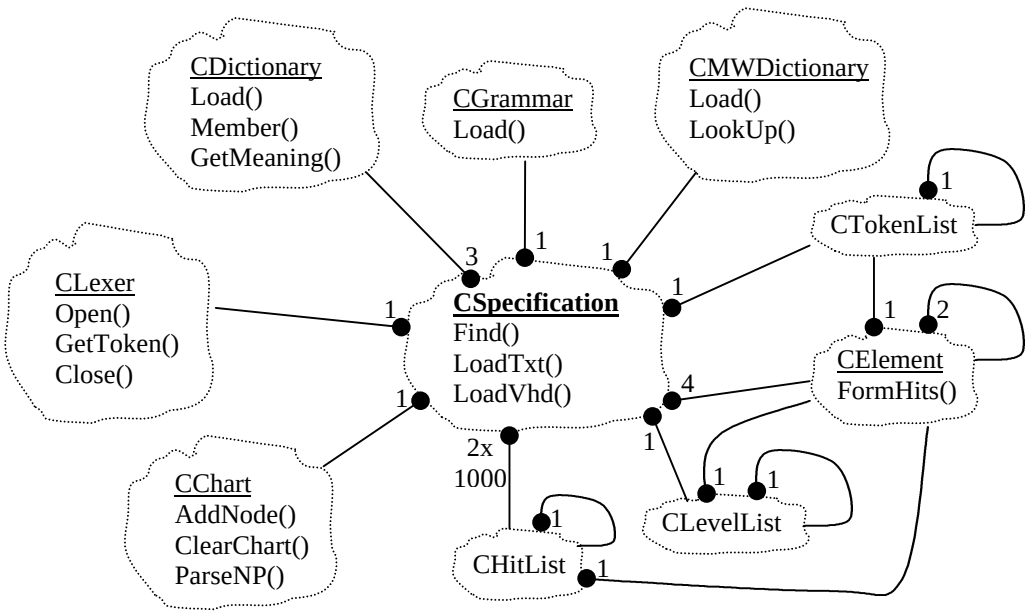


Figure 4-4: CSpecification Class Diagram

The CSpecification uses a CLexer object to obtain the specification document token. This task is primarily done by CSpecification::LoadTxt() (section 4.3.2.1). Each token that the CSpecification object receives from the CLexer object is then parsed for its hierarchy type. This is done to specify whether the token is a part of a paragraph, a section header or a table.

Each token that is a part of a sentence, is then passed to the CChart object by means of CChart::AddNode(). This allows the CChart object to create a new node for the token, and to parse it for possible new constituents. The CChart object is also responsible for supplying the CSpecification object with the list of noun phrases that it has recognized. These noun phrases will be used as the noun phrase search keywords.

The private member function CSpecification::LoadVhd() (section 4.3.2.1) is responsible for loading the VHDL input file to the CSpecification object. While CSpecification::LoadVhd() loads the input file, it also extracts all of the VHDL

identifiers. These identifiers are the words that will then be used by the CSpecification object as identifier keywords.

For efficiency, both the identifier keywords and the noun phrase keywords are sorted and filtered for duplicates. Once all noun phrases from the CChart object and the identifiers from the VHDL input file are compiled and filtered, the CSpecification object re-parses the document to search for occurrence of these keywords. The results of these searches are indexes stored in CHitList objects. A CHitList object is simply an object with two pointers and an integer. One pointer is pointing to the CElement objects that contain the noun phrase or identifier, and the other one is pointing to the next CHitList object. The integer indicates the number of occurrences (hits) that a keyword has. All of these processes and searches are done before the main window of ModelMaker is initialized. Pre-processing allows faster searches during the modeling activity.

4.3.1.2 CChart Class

The CChart class, shown in Figure 4-5, is responsible for building the parse tree from each of the parsed sentence. This class implements the Noun Phrase Extractor.

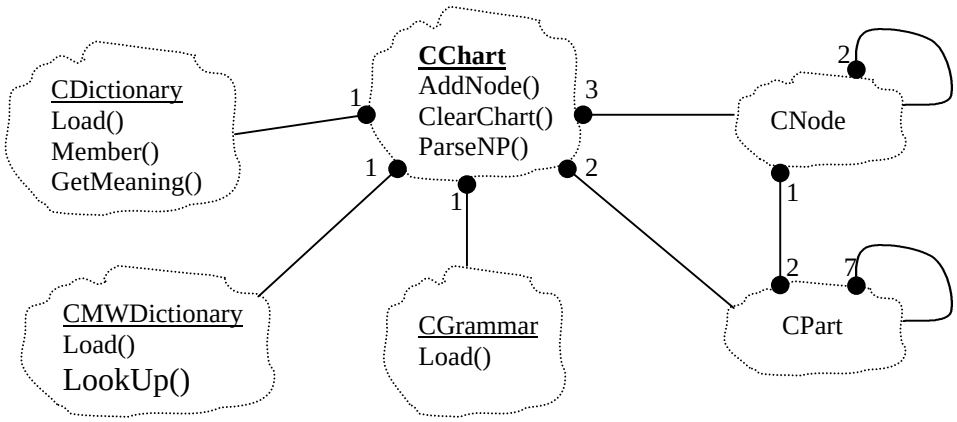


Figure 4-5: CChart Class Diagram

The CChart class uses the CNode class, CPart class, CDictionary class, CMWDictionary class, and the CGrammar class. A node is an object that carries information about one word, and a constituent (a CPart object) is a sequence of one or more nodes that forms a grammatical constituent according to the rules of the CGrammar object. Two pointers of CPart class and three pointers of CNode class are used to construct a CChart object. These pointers point to objects such as the first and the last node or constituent objects on the chart.

The links to the CDictionary, CMWDictionary, and CGrammar classes are also implemented as pointers and are initialized upon the instantiation of CChart object. The objects of these classes are used by CChart to store the dictionary entries, multi-word dictionary entries, and the grammar rules.

An object of CChart starts with empty pointers to the node and the constituent objects. For each word of a new sentence, a node object is created by CChart::AddNode() function (see section 4.3.2.2). Each time a node is added to the CChart data structure, a single constituent is also added. This constituent is a terminal constituent. The CGrammar object is then invoked to apply grammar rules that can combine this constituent with any other earlier constituents. If a rule is found, then a new CPart object is created and attached to the list of CPart objects for this node.

4.3.1.3 CDictionary Class

The CDictionary class, as represented in Figure 4-6, is responsible for creating and maintaining the dictionary data structure. Objects of this class are used to store the basic dictionary data, the structural dictionary data, and the behavioral dictionary data. Each of these dictionary objects has a private pointer of type CEntry that holds the first entry of the dictionary. Moreover, each of these CEntry objects has a pointer to the next entry of the dictionary. These structures create link-list data structures with the first entry of each dictionary as their roots. Each CEntry object contains a vocabulary word, and each CMeaning object contains a category.

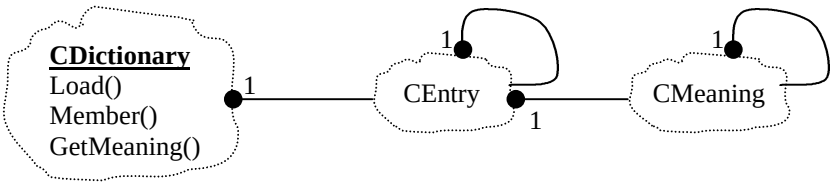


Figure 4-6: CDictionary Class Diagram

The CDictionary object is initialized as soon as it is created. The constructor of this CDictionary object requires the file name of the dictionary file as an argument. This constructor is the one that will call the private CDictionary::Load() member function (section 4.3.2.3) to load the dictionary file. The only access given to the CDictionary object user is the public member function CDictionary::Member() and CDictionary::GetMeaning(). This limitation ensures that there is no sharing violation between CDictionary objects.

To speed up the search process for any words, a two-dimensional pointer array of CEntry type is used as a hash table. The first dimension is used to index the first character of the words, and the second dimension is used to index the second character of the words. This hash table is needed as the dictionaries are getting larger and larger (the current dictionary has more than 4000 entries).

4.3.1.4 CMWDdictionary Class

The class diagram of the CMWDictionary class is presented in Figure 4-7. The CMWDictionary class is similar to the CDictionary class in its function. As the CDictionary object creates and holds the data of standard dictionaries, a CMWDictionary object creates and holds the data of the multi-word dictionary. Its data structure, however, uses the constant array structure instead of the link-list structure. This choice is based on the same reasoning with the choice of data structure for the CGrammar class. As the number of entries for the CMWDictionary object is expected to be quite small and

constant, the constant array structure provides more efficiency. The current limit for this array is 100 entries.

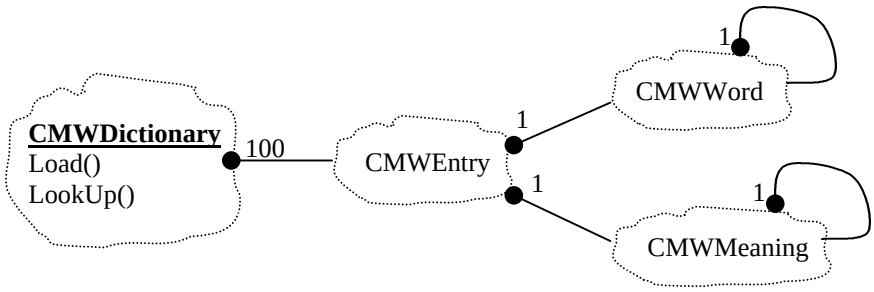


Figure 4-7: CMWDictionary Class Diagram

4.3.1.5 CGrammar Class

As the CDictionary class is responsible for creating and holding the dictionary objects, the CGrammar class, shown in Figure 4-8, is responsible for creating and holding the grammar objects. The data structure of the grammar entries is implemented in a constant array of objects. This fact limits the current CGrammar object, as it can only hold up to 150 rules (this limit can be increased simply by changing a variable in the code). The grammar size is expected to be moderately constant, and the access speed of a constant array structure is faster than the access speed of a linked-list structure.

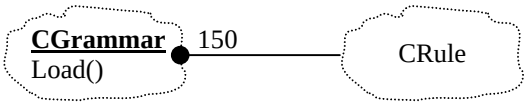


Figure 4-8: CGrammar Class Diagram

The CGrammar::Load() member function loads the dictionary object from an external file. This function needs to be called explicitly as an automatic loader is not yet implemented in the CGrammar constructor.

To access any of the rules, the CGrammar user can directly call the CRule object of the grammar structure by its index. An integer public-member-variable of this CGrammar, *nRules*, is used to store the number of grammar rules that are successfully loaded by this class. The CGrammar user can use this number as the upper limit in accessing the grammar rules.

4.3.1.6 CElement Class

CElement objects are used to store the source document inside the CSpecification object. The CElement class is a generic class that can be used for section, paragraph, and sentence elements. An element object will have a list of tokens, a list of constituents (other element objects below the level of this element object), and a list of other elements of the same level. Figure 4-9 presents the class diagram for the CElement class.

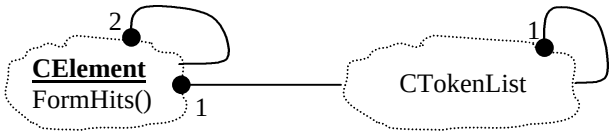


Figure 4-9: CElement Class Diagram

The list of token objects, of CTokenList class, is used to store the section headers for the section elements and the sentences for the sentence elements. The CElement object uses two pointers of the same class for the list of constituents and list of other elements. Sentences are constituents of paragraph; and paragraphs are constituents of section. The list of other elements pointer is used to point to the next element of the same level.

The CElement class also keeps three integers, m_Hits, m_BehaviorHits, and m_StructurHits, to count the number of matched keyword, behavioral word or structural word that is being searched. These integers are used by the CElement::FormHits() functions. More information about these integers is presented in the FormHits() function discussion.

4.3.1.7 CLexer Class

The CLexer class is a small but essential class. As shown in Figure 4-10, this class does not use any object from other classes. The only task of an object of this class is to read the input file, character by character, and compile a token to be given back to its user. A token can be a word, number, or punctuation. This class distinguishes a period at the end of a sentence from a decimal point in a number. Using this CLexer class, the string "sentence," will be separated as "sentence" and ",". Also, the section header "1.1.3" will be kept as one word instead of "1", ".", "1", ".", and "3".

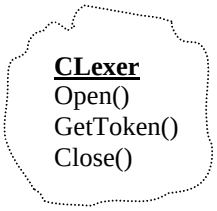


Figure 4-10: CLexer Class Diagram

To define the term *token* in more detail, the rules that govern ModelMaker in demarking tokens are listed below.

- A ***Token*** can be formed from either
  - *Letters* or
  - a *Number* or
  - a *Punctuation* or
  - a *Special punctuation* or
  - a *Blank space*

- ***Letters*** can be formed from either
  - a *Letter* followed by *Letters* or
  - a *Letter* followed by a *Number* or
  - a *Period* followed by *Letters* or
  - a *Hyphen*, "-", or
  - a *Slash*,"/", or
  - an *Underscore*, "_", or
  - any letter between *A* to *Z* or
  - any letter between *a* to *z*

- A ***Number*** can be formed from either
  - a *digit* followed by a *Number* or
  - a *Number* followed by a *Letter* or
  - a *Special punctuation* followed by a *Number* or
  - any single digit number between *0* to *9*

- A ***Punctuation*** can be formed from either
  - a *Colon*, "*:*", or
  - an *Exclamation mark*, "*!*", or
  - a *Left curly bracket*, "*{*", or
  - a *Left parenthesis*, "*(*", or
  - a *Left square bracket*, "*[*", or
  - a *Question mark*, "*?*", or
  - a *Right curly bracket*, "*}*", or
  - a *Right parenthesis*, "*)*", or
  - a *Right square bracket*, "*]*", or
  - a *Semi-colon*, "*;*"

- A ***Special*** punctuation can be formed from either
  - a *Period*, "*.*", or
  - a *Comma*, "*,*"

- Any other character string is recognized as a ***Blank***

4.3.2 Main Member Functions

As each ModelMaker class has many member functions, not all of them will be discussed in this section. Only member functions that are crucial for the discussion of ModelMaker intercommunication is presented here.

4.3.2.1 Main Member Functions of CSpecification Class

The main member functions of the CSpecification class include LoadTxt(), LoadVhd(), and Find(). This section discusses the functionality and the internal behavior of these functions.

LoadTxt(FileName, aChart). This private member function requires two arguments, the name of the source document file and the pointer to the CChart object. This function scans the source document file, constructs the document inside the m_pSpec object, and communicates with the CChart object. Communication to the CChart object is done by calling the CChart::AddNode(), CChart::ParseNP() and CChart::ClearChart() functions. This communication is needed to let the CChart object create sentence parse trees. The CSpecification::LoadTxt() function is called by the CSpecification constructor. The input for this function is the token returned from the CLexer object. This function returns a flag to indicate whether it was run successfully or not.

A document inside the m_pSpec object is a collection of document elements. These elements include sections, paragraphs, sentences, tables and figures. To maintain the document hierarchy, these document elements are built from the CElement class. In the current implementation, a section leads paragraphs, and a paragraph leads sentences, tables, and figures. The current algorithm implemented in this LoadTxt() function tries to recognize the document elements automatically for any input document. A set of general rules to format a document for recognition is described in details in section 4.2.4.2. These rules are quite general and should include many document formats without requiring any adjustment.

The following two figures show the CSpecification document hierarchy built by CSpecification::LoadTxt(). Figure 4-11 shows a sample document; and Figure 4-12 shows how this document will be formed into a CSpecification object.

1 FUNCTIONAL DESCRIPTION

The 8237a block diagram includes the major logic blocks and all of the internal registers. The data interconnections paths are also shown.
...

Figure 3 8237A Internal Registers:

Name:	Size:	Number:
Base Address Registers	16 bits	4
Base Word Count Registers	16 bits	4

...

2 DMA OPERATION

The 8237A is designed to operate in two major cycles. These are called the Idle and Active cycles.
...
.
.
.

Figure 4-11: Sample Specification Document Text (partial)

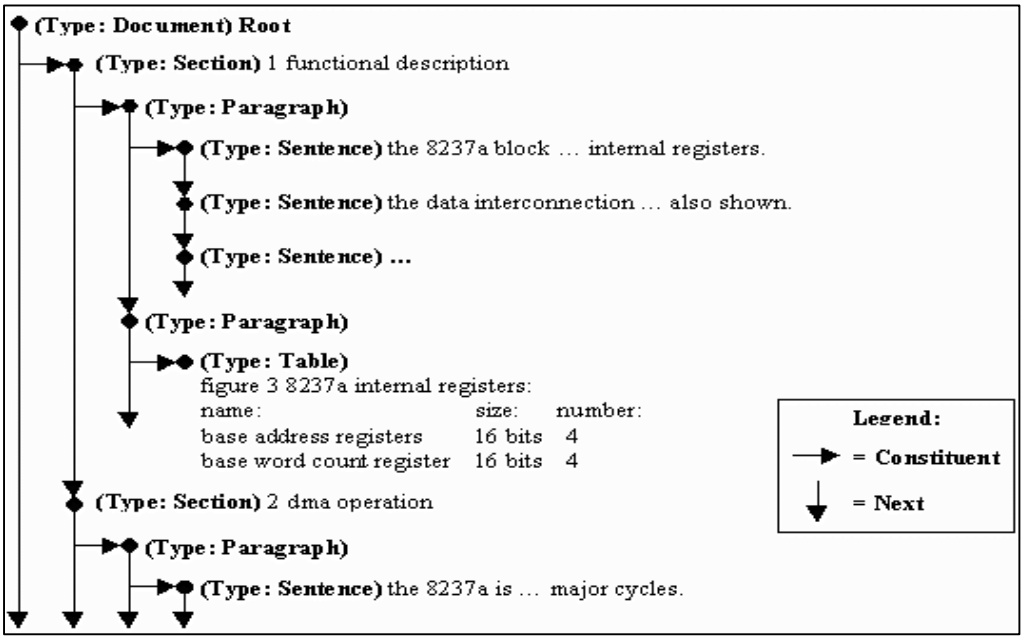


Figure 4-12: Document Elements Built by CSpecification::LoadTxt()

The CChart::AddNode() function is called every time a new token is received from the CLexer object. This action is needed to let the CChart object create the parse

tree for the current sentence. Refer to section 4.3.2.2 for more description on the CChart class member functions.

The CChart::ParseNP() function is needed to extract all of the noun phrases that exist in the current parse tree. This function is called every time the CSpecification object finds a new section header or a new paragraph. The return value of the CChart::ParseNP() function will then be attached to the noun phrase index list in the CSpecification object. Once the end of the input file is reached, this noun phrase index list will be sorted and filtered for duplicates. The actual search for this index list will be done by the CSpecification constructor right after the CSpecification::LoadTxt() and the CSpecification::LoadVhd() member functions are executed.

As soon as CChart::ParseNP() is done with its task, the CSpecification object calls the CChart::ClearChart() function. This is done to remove and reinitialize the entire chart.

LoadVhd(FileName). This private member function requires one argument, the name of the VHDL model file to be loaded. Similar to LoadTxt() function, this function is also an essential function in initializing the CSpecification class. The main task of CSpecification::LoadVhd() function is to load the VHDL input file (see section 4.2.4.1) into the CSpecification document object. Another important task of this function is to parse the VHDL input file for identifiers. These identifiers are used in the identifier index list after they are sorted and filtered for duplicates. This function returns a flag to indicate whether it was run successfully or not.

Find(Keyword, And_Or, Id_Np). This public member function requires three arguments, the keyword, the search type, and the keyword type. The function provides a way for the caller object to search the specification document for a specific keyword. This function can perform Boolean “AND” and “OR” searches on multiple word keys, depending on the second argument. The third argument specifies whether the keyword is a VHDL identifier or a noun phrase. A public integer variable of the CSpecification class,

m_Mode, defines the sorting mode. The caller object needs to set this variable to sort the results *sequentially*, *behaviorally*, *structurally*, or *relevantly*. All of these search types and modes will be discussed in more details in the Keyword Searches section (section 4.4). In the current implementation of ModelMaker, this Find() function is the main engine behind the "Do Search" button. The return value of this function can be sent directly to the Source Window of ModelMaker.

Since the occurrences of all keywords are searched on the initialization of ModelMaker, CSpecification::Find() can find the occurrence of any keyword very quickly. All the function needs to do is get the results, based on an index, and to return them. This implementation allows ModelMaker to quickly respond to the modeler queries. One disadvantage of this method is a slow initialization time for ModelMaker.

ModelMaker also allows the modeler to enter new search terms, by typing in the keyword window or selecting a particular term inside the Model Window. A new search is conducted for each new keyword that is not in the list-of-keyword index. This new result will then be added to the database, with its keyword added to the keyword index, for future searches.

Based on the "Search Type" desired by the modeler, the search results of a multi-word keyword are combined. Similarly, based on the "Search Mode" desired by the modeler, the final search results are sorted before they are returned to the modeler. The combining and the sorting processed will be discussed in more detail in the Keyword Searches section (see section 4.4).

4.3.2.2 Main Member Functions of CChart Class

There are four important member functions in CChart class, they are AddNode(), ParseNP(), ParseSentences(), and ClearChart(). The functionality and internal behavior of these functions are discussed in this section.

AddNode(m_pToken). The AddNode() public member function is needed to add a node to the CChart parse tree. This function takes one argument, the *token* to be added as the new node. When the new node is the first node in the chart, this member function will also create and initialize the chart. If the chart exists, a new node is created and added to the end of the node list of the existing chart. This function returns a flag to specify whether it was run successfully.

Once the node is created, the GetMeaning() member function of the standard CDictionary object is called to determine the token's category (part of speech). If one of the parts of speech of the token is an asterisk sign, however, the token may be a part of a multi-word token. In this case, the multi-word dictionary object is consulted. This can be done using the CMWDictionary::LookUp() public member function. For each category, either from the standard dictionary or from the multi-word dictionary, a new constituent is created and attached to the node object for this particular token. These constituents are called the basic constituents as their categories are taken from the dictionary. Constituents are stored in CPart objects in the chart.

If the category of a word is not found, either in the standard dictionary object or in the multi-word dictionary object, a category called *identifier* will be assigned for that word. This is done to handle acronyms and identifiers such as device names and device types. One disadvantage of this implementation is that every unknown word from the specification document will be recognized as an *identifier*, which acts as a noun. To avoid misclassification of new verbs, the dictionary files should be updated for new specifications. To help this process, ModelMaker source files can be recompiled with its DEBUG definition set to "1." When this definition is set, ModelMaker will keep a record of unknown words that it encounters. This record will then be saved in the text file called "unknown.txt" as soon as this application is terminated. The user should review this file with new documents.

Once all of the basic constituents for a particular node are built, the CChart::AddNode() function searches for any possible new constituents that can be

constructed with the grammar rules. If a new constituent can be formed, it will be added to the node's list of constituents. This will continue until no more new constituents can be built from the existing constituents.

ParseNP(). No argument is required for this public member function. This function is called just before the CChart::ClearChart() function is called. The purpose of this function is to scan the CChart object for noun phrase constituents. For each noun phrase found, a noun phrase identifier is created and attached to the noun phrase identifier list. To keep the identifiers short, the words *a*, *all*, *an*, *each*, *that*, *the*, *these*, and *this* are deleted from them.

This function returns all noun phrases in a single CString object. In this CString object, all the white space characters are replaced by underscore signs, "_", and a pipe sign, "|", is placed at the end of each noun phrase.

ParseSentences(). This public member function has no argument and no return value, and is invoked to scan a chart and to find all of sentence constituents in the existing chart. For each sentence constituent, a parse tree is constructed, converted to a Prolog structure format and written to the file "suite.pl" for analysis by the Template Constructor.

ClearChart(Capture). This public member function requires a Boolean argument, *Capture*. Since ModelMaker is intended to read a long specification document with many sentences, maintaining a complete chart for the whole specification document is not practical. Each node can have as few as one constituent and as many as a thousand or more constituents. The span of each of these constituents can be as small as a single word or as long as the whole sentence. This function provides a way for the CSpecification object to delete all node and constituent objects when it detects the beginning of a new section or paragraph or the end of a sentence or the document. If the value of Capture is *true*, the existing chart is returned as a CString object. Otherwise, the return value of this function is an empty CString object.

While doing the cleaning task, the ClearChart() function also searches the existing chart for any constituents representing a complete sentence, by means of CChart::ParseSentences() function. For each of such sentence constituents, a parse tree is constructed and added to the Template Constructor input file, "suite.pl". These trees will be analyzed by the Template Constructor.

4.3.2.3 Main Member Functions of CDictionary Class

There are three main member functions of CDictionary class that are presented in this section. These functions are the Load(), Member(), and GetMeaning() functions.

Load(FileName). This private member function is used to load a dictionary input file for constructing a CDictionary object. A file name is required as its argument. This function is a private function, since the only place it is used is in the CDictionary class constructor. A flag is returned by this function to specify whether it was run successfully.

This function is an enhanced version of the CDictionary::Load() member function used in the ASPIN Parser. The main enhancements for ModelMaker include better error detection and error handling, unlimited dictionary entries, and variable length words. A new algorithm avoids various types of error in dictionary formatting.

Member(Word). An argument, *Word*, is needed for this public member function. This function provides a quick way for the user to check whether a token is listed in the dictionary. The argument specifies the token to be searched. If the token is a member of the dictionary, the function will return a CEntry pointer that points to the CEntry object for that token. Otherwise, a *null* pointer is returned to the caller object. A 2-character index table is used to accelerate the search process.

GetMeaning(Word). This GetMeaning() public member function is used to obtain the category, or part of speech, for a particular token stored in the dictionary object. Similar to the Member() function, an argument is needed for this function. This argument

specifies the token to be searched. Internally, this function uses the Member() function to obtain the CEntry object of the token in question. If the token is a member of the dictionary, the function will return a CMeaning pointer that points to the CMeaning object for that token. Otherwise, a *null* pointer is returned to the calling object.

4.3.2.4 Main Member Functions of CMWDictionary Class

There are two main member functions in the CMWDictionary class. These member functions are the Load() and LookUp() functions.

Load(). No argument is needed for this public member function. This function loads a multi-word dictionary input file, "mdict.txt", for a CMWDictionary object. A flag is returned to the caller object to specify whether it was run successfully. The data structure used in this CMWDictionary class is a combination between the data structure of the CDictionary class and the CGrammar class. Its division of entries and categories follows the CDictionary class style, but, fixed length variables are used store the data, as in the CGrammar class. This implementation was chosen since the number of entries for a multi-word dictionary is expected to be a small constant. For the current implementation, the data structure can only store up to 100 entries. Another limitation on the current implementation is the maximum number of characters in each word, each category, and each concept. These numbers are set to 19, 8, and 19 respectively. All of these limitations can be easily modified by changing the numbers directly in the CMWDictionary source code.

LookUp(pNode). The argument needed for this public member function is the CNode object of the last word in the multi-word in question. This public member function provides a simple way for the user to access the categories of a particular multi-word. LookUp() is performed by a simple linear search. If an entry is found for the keyword that the user searched for, a pointer to the matching CMWEntry object will be returned. Otherwise, a *null* pointer is returned.

4.3.2.5 Main Member Function of CGrammar Class

The only member function of the CGrammar class is the Load() function. To access the rules of a grammar object, the user can simply refer to the public-member-variable array of CRule objects used to store the grammar.

Load(). This public member function is similar to the CDictionary::Load() function. This function loads the grammar file, "gram.txt", to construct a CGrammar object. In the current implementation, the CGrammar data structure can only store up to 150 grammar rules. If the grammar file is loaded successfully, the number of grammar rules loaded is returned to the caller. The function requires no argument.

4.3.2.6 Main Member Functions of CElement Class

The CElement class is a generic class used to create the sentences, paragraphs, sections, and documents element. This function only has one main member function, the FormHits() function.

FormHits(Token, Hits, BehaveHits, StructHits, pFirstHit, aBehaviorDic, aStructuralDic, CheckBehavior, CheckStructural). This public member function is a recursive function used to find all occurrences of keywords in the document elements. This function requires nine arguments. The first one, *Token*, specifies the token to be searched. *Hits*, *BehaveHits*, and *StructHits* are call-by-reference arguments [24]. These arguments are used to pass the number of hits (matched tokens) from the last level to the first level of the recursion. The values of these arguments need to be reset to zero as they are passed to the function. The fourth argument, *pFirstHit*, points to the first matched element. The fifth and sixth arguments are pointers to the behavioral and structural dictionary objects. Finally, the last two arguments are Boolean flags to specify whether behavioral and structural vocabulary checks need to be performed. A flag is returned by this function to specify whether it was run successfully.

For each of the keywords, noun phrases and identifiers, the m_pSpec object calls this function during its initialization. This is done to prepare the search results index.

4.3.2.7 Main Member Functions of CLexer Class

GetToken(). This public member function is the main member function in the CLexer class. When called, this function returns the next token from the open document file in the form of a character pointer. If the end of file is reached or any error is detected, a *null* pointer will be returned. In this implementation of CLexer class, all tokens will be returned in lower case letters. No argument is needed in calling this GetToken() function.

4.4 Keyword Searches

To explain the basic design of ModelMaker, some of its main algorithms are discussed here. Since the occurrences of the keywords are pre-searched during ModelMaker initialization, most of the discussions in this section deal with how ModelMaker returns this search result back to the modeler. The pre-search is done by the CSpecification class.

The pre-search is performed just after the LoadVhd() and LoadTxt() functions execute. Pre-search fills the results index with the search results (hit lists) of the keywords in the keywords index. For each of the keywords, the CElement::FormHits() function (refer to section 4.3.2.6) is called using the keyword tokens as the search keys. This is done for both the VHDL identifier keywords index and the noun phrase keywords index. For the VHDL identifiers, the pre-search processing is straightforward as they are single tokens. Most of the noun phrases, however, are composed of two or more tokens. Each word from these multiword noun phrases needs to be searched separately. The results are then combined.

4.4.1 AND-Search-Mode

The AND-search-mode specifies how ModelMaker builds the hit lists for multiple token keyword searches. In this mode, ModelMaker searches each paragraph for any occurrence of each tokens in the keyword. Anytime a match is found in a paragraph, the section header of the paragraph is added to the search result. The sentences in that paragraph, however, will not be added to the search result unless all of the words in the keywords occur in that particular sentence. In some cases, the section header will be displayed without any sentences written under it. This is just to show that one or more, but not all, of the tokens in the keywords occur under this section. By reducing the number of tokens in the keyword, the modeler may be able to find more results under this section header.

4.4.2 OR-Search-Mode

Similar to AND-search-mode, OR-search-mode specifies how ModelMaker builds the search result of multi-word keyword searches. In this mode, the search result from each of the word in the keywords is displayed separately under each of the keywords. This feature allows the modeler to see the search result of each of the words independently.

This OR-search-mode is only available for the identifiers, or user typed, keywords. This limitation was established since the noun phrase index list is supposed to contain the entire noun phrase found by ModelMaker.

4.4.3 Sequential-Sort

The sequential sort mode is the default mode for any ModelMaker search. In this mode, search result is sorted based on its original order in the source document text. All section headers will also be displayed to the modeler. This mode is very useful for modelers to find where that particular instance is in the actual document specification

text. This allows modelers to go back to the actual document specification text for more detailed information.

4.4.4 Relevance-Sort

Relevance-sort mode sorts the search results based on the number of keywords (from the entire keyword list) that occur in the paragraph. The more keywords that occur in the paragraph the higher its relevancy. As the order of sentences in this mode may be different from the actual order in the source document text, the section headers for these sentences are not displayed. This mode is useful for modelers in finding any information that is more general to all the identifiers and the noun phrase. This sort returns the same information returned by the other sorts. The only difference is in the order of the results.

4.4.5 Behavioral-Sort

In this behavioral sort mode the search result is sorted based on the number of behavioral words that each paragraphs has. An object from the CDictionary class is used to hold the behavioral word dictionary. While performing the initial search, ModelMaker checks for any occurrence of behavioral words. The more behavioral words the paragraph has, the higher its behavioral content. Based on a similar reasoning with the relevance-sort, the section headers for these paragraphs are also not displayed. As behavioral words are usually used in describing the behavior of a device, this sorting mode is useful in finding more information about the device. The behavioral word dictionary is saved in the “Behavior.dic” file. This sort returns the same information returned by the other sorts. The only difference is in the order of the results.

4.4.6 Structural-Sort

Similar to the behavioral-sort, this mode is also based on a CDictionary object, the structural dictionary. The structural word dictionary is saved in the “structure.dic” file. In this mode, a paragraph that has more structural words on it will have a higher

structural content, which will be displayed first. ModelMaker also parses the input specification document for these structural words during its initial search. Section headers for these paragraphs are also not displayed. This search mode is useful to the modelers in building the structure of the device. This sort returns the same information returned by the other sorts. The only difference is in the order of the results.

CHAPTER 5: ASPIN COMPONENTS

5.1 Noun Phrase Extractor (Parser)

The Noun Phrase Extractor is an adaptation of the ASPIN Parser. The ASPIN Parser is a stand-alone console application written in C++ object-oriented programming. The Parser is supported by a context-free grammar and a dictionary giving the lexical categories of words.

To adapt the Parser, some of the classes were rebuilt and modified to make them more compatible to the classes of ModelMaker. The initial code, which could handle only a relatively small input file, was modified to allow very large input files. Some other additions and enhancements were also done to the Parser classes during this integration process.

5.1.1 ASPIN Parser

The ASPIN Parser is a bottom-up, knowledge-based sentence parser that constructs parse trees for input sentences. The knowledge bases used by the Parser are a dictionary database and a grammar. The following paragraphs contain a simplified discussion of the Parser.

The dictionary contains lexical categories for about 1200 vocabulary words. The second database, the grammar database, contains about 120 context-free grammar rules using about 48 non-terminals. A non-terminal is a grammatical constituent that can be separated into two or more smaller constituents or lexical categories. The Lexical categories are the terminals. The terminals are *adjective*, *adverb*, *conjunction*, *determiner*, *identifier*, *modal*, *negation*, *noun*, *time order*, *pre-determiner*, *preposition*, *pronoun*, *relative conjunction*, *subordinating conjunction*, *past participle*, *verb*, *infinitive verb*, *present participle*, *infinitive be*, *be*, *to*, *of*, and *have*. Examples of non-terminals are noun phrase, sentence, verb sequence, subordinating clause and predicate. Parsing is

performed by constructing a chart for each sentence. The chart includes a node for each word of the sentence. Each node has at least one constituent. This constituent is the category of the word. The category for the first constituent is obtained from the dictionary. Once this constituent is added successfully, the grammar rules are applied. All rules are considered to determine if the most recent constituent can be combined with any constituents of previously analyzed nodes to form new constituents. This process is repeated until there are no more rules that can be used to combine any of the existing constituents.

The primary classes of the Parser are CChart, CDictionary, and CGrammar. The Parser employs a CParser class to use these classes. In ModelMaker, however, this task is performed by the CSpecification class. For more detailed information on how the initial version of the ASPIN Parser works, consult [12].

5.1.2 Enhancements and Adaptations

Many enhancements and adaptations were needed in creating the Noun Phrase Extractor from the ASPIN Parser. Some of these enhancements include a memory-leak checker, a multi-word recognition ability, a rule-skipping capability, and output constituent selector. Some of the adaptations add robustness and error detection. A new dictionary class was implemented with an enhanced search algorithm.

The memory-leak checker was implemented to avoid any memory leaks. Memory leaks, which can crash an application or even an operating system, are usually caused by pointers that are not managed properly. Memory leaks create an unstable system, which endangers other applications that are running on the system. The memory-leak checker is implemented as a simple counter that holds the number of objects of a particular class at any time. Each time an object of a particular class is created, the counter for that particular class is increased. Similarly, each time an object of a particular class is destroyed, the counter for that particular class is decreased. If the counter of a particular class is not equal to zero at the end of the lifetime for the particular class, one

can predict that the application contains potential memory leaks. If this counter goes below zero, one or more destructors[5] have been called one too many times. When the memory-leak checker finds either of these two situations, it will warn the user through a pop-up dialog box. Since this process added more complexity to the class constructor and the class destructor, this feature is activated only in the debugging code and not the release executable. Using this feature, most of the memory leaks in the Noun Phrase Extractor can be eliminated.

In the attempt to create a more stable application, some of the classes of the Noun Phrase Extractor were reimplemented. These new classes use the concepts of private and public members. This implementation enforces the use of object-oriented rules for interclass communications. This is needed to ensure that the code is maintainable and modifiable.

Another enhancement for the Noun Phrase Extractor includes the multi-word recognition capability. Using a multi-word dictionary, the Noun Phrase Extractor can recognize multi-words such as “greater than” and “consist of” as terms having a lexical category. The last word from each of the multi-words carries a special category flag in the standard dictionary. Each time a word with this flag is found by the Noun Phrase Extractor in the standard dictionary, it will check whether this word, together with the earlier words, can be combined as a multi-word as well.

In the new Noun Phrase Extractor, a rule-skipping capability is also provided. This capability allows the grammar rules to skip one or more words in matching constituents. By utilizing this capability, the Noun Phrase Extractor can skip words to recognize more sentences. However, this ability increases the processing time. Therefore, this feature is disabled in the current version of ModelMaker.

[5] A destructor is a set of code that is supposed to dereference and to clean all of the memory used by a class.

One of the most important additions to the Noun Phrase Extractor is the noun phrase constituent selector. The selector scans the chart and finds all noun phrase type constituents. All of the noun phrases found by this algorithm are then passed to ModelMaker to create its noun phrase index.

The current standard dictionary contains over 4000 words and the multi-word dictionary contains 21 entries.

5.2 Template Constructor (Semantic Analyzer)

The Template Constructor is an adaptation of the ASPIN Semantic Analyzer. The primary function of the Semantic Analyzer is to create conceptual graphs representing the meaning of the sentences in the input file. The Template Constructor is an external component used by the ModelMaker to perform semantic analysis on sentences and fill behavioral templates from them. The Template Constructor is implemented as a set of Prolog rules.[6] Whenever the Template Constructor is needed by the ModelMaker tool, a system-call to the Prolog interpreter is executed to interpret the Template Constructor rules. ModelMaker writes to an input file, in prolog-rule style. The Prolog interpreter will then load this input file, interpret the rules, and write the results to an output file. This is not the most efficient way to do the communication, but it is sufficient at this point because the primary objective of this research is to study the feasibility of the ModelMaker tool as a system.

An example of the Template Constructor output is shown in Figure 5-2. This result is based on the parse tree shown in Figure 5-1. The parse tree is taken from the chart of the Noun Phrase Extractor.

[6] Prolog is an "interpreted" language (refer to Programming in Prolog [7]). Prolog source code contains a list of rules instead of program code. Prolog uses these rules as a knowledge database. A Prolog interpreter is always needed to interpret Prolog rules, because it can not be compiled to binary format.

```
=== Sentence #330
the device will output an hrq to the microprocessor and enter the
active cycle .

   s(330,c(
    s2,[c(
     sp2,[c(
      n1,[c(
       np12,[t('det','the'),c(
        head1,[t('noun','device')])])]),c(
     pred3,[c(
      avs3,[t('mod','will'),t('vinf','output')]),c(
       n1,[c(
        np12,[t('det','an'),c(
         head2,[t('id','hrq')])])]),c(
          d2,[t('prep','to'),c(
           n1,[c(
            np12,[t('det','the'),c(
             head1,[t('noun','microprocessor')])])])])])]),
     t('conj','and'),c(
      pred2,[c(
     avs2,[t('verb','enter')]),c(
      n1,[c(
       np11,[t('det','the'),c(
        adjs1,[t('adj','active')]),c(
         head1,[t('noun','cycle')])])])])]),
     t('.','.')])).
```

Figure 5-1: Sample Parse Tree

=== Sentence #330
the device will output an hrq to the microprocessor and enter the active cycle .

1 AND < *behavior*
  and: **ENTER** (2)
  and: **OUTPUT** (3)
2 ENTER < *call* < *event*
  enters: **CYCLE** (4)
  agent: **DEVICE**, the device
4 CYCLE < *state* < *behavior*
  attribute: **ACTIVE**
3 OUTPUT < *move* < *action*
  destination: **MICROPROCESSOR** < *processor* < *device*, the microprocessor
  operand: **HRQ** < *id*, an hrq
  agent: **DEVICE**, the device
  modal: **WILL**

Figure 5-2: Sample Result of the Template Constructor

The template shown in Figure 5-2 is based on general templates. The general template for action type concepts is shown in Figure 5-3.

action: *an action*
- **agent**: *the device that performs the action*
- **destination**: *the device receiving data from the action*
- **source**: *the device supplying data to the action*
- **instrument**: *a device or carrier that is used by the action*
- **operand**: *an input value to the action*
- **result**: *an output value produced by the action*
- **if**: *a state or condition that enables the action*
- **causes**: *an action caused by the behavior*
- **generates**: *an event generated by the action*
- **initiated**: *the event that initiates the action*
- **begin**: *the event that begins the action*
- **end**: *the event that ends the action*
- **manner**: *a description of how the action is performed*
- **temporal**: *a temporal relation with the occurrence of another action or event*

Figure 5-3: General Template for Action

The example shows the Template Constructor results on the sentence "The device will output an HRQ to the microprocessor and enters the active cycle." In the resulting template, the bold words are the concepts represented in the sentences. The modeler can observe that the *output* action is performed by *the device*. Similarly, the input value is the *HRQ*, and the destination is *the microprocessor*. The "<" symbol indicates an "is a" relationship; and the indented items show the relations between concepts.

5.2.1 ASPIN Semantic Analyzer

The initial version was implemented in Quintus Prolog to run on Sun workstations. It accepts parse trees as Prolog structures and generates semantic analyses in the form of conceptual graphs [8,9,20], a form of semantic networks.

To invoke the Semantic Analyzer, the main file, "main.pl," needs to be called from the Quintus Prolog environment. This file loads the knowledge database files and

the input file. The rule-database files are the root database, the semantic rule database, and the canonical graph database. The root database contains information about the root word of each word listed in the dictionary. This information is used by the Semantic Analyzer to determine the root word for each word found in the sentence. The semantic rule database contains the semantic rules used by the Semantic Analyzer to guide the attachment of canonical graphs. These rules are based on the grammatical structures of the ASPIN Parser application. The canonical graph database contains the canonical graphs for words that are currently covered by the Semantic Analyzer vocabulary. Finally, the input file needed by this application is the "suite.pl" file. This file contains lines of Prolog rules, with each line representing a parse tree of a sentence. These parse trees are generated manually from the result of the ASPIN Parser application.

For more information about the initial version of the Template Constructor, see [12].

5.2.2 Enhancement and Adaptations

The Semantic Analyzer was enhanced as part of another effort[7] to deliver human-readable templates instead of conceptual graphs. The Semantic Analyzer is called the Template Constructor in the ModelMaker tool. Because Windows 95 is the target platform for the Template Constructor, the first adaptation to the Semantic Analyzer was converting some of the Quintus Prolog specific rules to SWI Prolog rules [22]. The public domain SWI Prolog interpreter was chosen because this interpreter is available in both the Unix platform and the Windows platform.

The Template Constructor input and output files were reformatted to the need of the ModelMaker application. Unlike its parent application, this Template Constructor only returns results when sentence analysis is successful. Because the results of the Semantic Analyzer were in a conceptual graph format, some adjustment was done in the

[7] NSF Grant MIPS-9707317.

Template Constructor to return the results in a template form.[8] An example of this result was presented in Figure 5-2.

This formatting includes the output layout, boldface, italics, and color. This formatting is attached to the output of the Template Constructor as markup tags. These markup tags are written in an HTML-like style. The current ModelMaker implements only three types of markup; they are bold, italic, and color. The markup rules used by ModelMaker are shown below.

- Bold = This will be printed in Bold
- Italic = This will be printed in Italic
- Color = <c:XXXXXX>This will be printed in XXXXXX Color</c>

In the *color* markup (XXXXXX), the first two-digit stands for hexadecimal digits (00 to FF) of the blue intensity. The second two-digit stands for hexadecimal digits of the green intensity, and the last two-digit stands for hexadecimal digits of the red intensity.

5.2.3 Template Constructor and ModelMaker Communications

The Template Constructor application and the ModelMaker application use an input file and an output file to communicate with each other. As soon as the ModelMaker tool acquires the chart of all sentences found in the specification document file, by means of the Noun Phrase Extractor, it creates the input file for the Template Constructor. This file is "suite.pl" file. Once this file is created, ModelMaker will also create the protocol file. This protocol file, "_runsemanal.pl," is a Prolog file that is needed to control the Template Constructor component. This protocol file contains the rules to redirect the output, the rules to invoke the Template Constructor, and the rules to analyze each of the input sentences. The latter is needed, because the Template Extractor component can only analyze one sentence at a time.

[8] Part of the development of this template-look form was done by Jeff Hess (Virginia Tech, Electrical and Computer Engineering Department). More information on this template-look form can be found in his

As soon as the modeler presses the "Template-Re-Analyze" button, a system call to the SWI Prolog interpreter is executed. This system call is done using the ShellExecute command in the Visual C++ language. The protocol file, "_runsemanal.pl," is passed to the Prolog interpreter as its argument. This protocol file will then initialize all of the knowledge databases and analyze each of the input files. This protocol also redirects the outputs of the Template Constructor to the "_out.txt" file. Once all of the sentences are analyzed, this output file will be closed by the Template Constructor. As soon as ModelMaker sees that this output file is ready, it reopens the file, displays the contents to the Source Window, and reformats the display based on the markup tags.

Master's thesis (in progress). This Master's thesis is expected to be published by December 1998.

CHAPTER 6: CONCLUSIONS

6.1 System Capabilities

The primary objective of this research was to create an Electronic Design Automation prototype application that combines the functionality of the ASPIN Parser and the ASPIN Semantic Analyzer, and adds significant new functionality. The purpose of ModelMaker is to help hardware designers improve their design productivity.

In the current version of ModelMaker, an enhanced version of the ASPIN Parser (Noun Phrase Extractor) application is tightly integrated in its binary code. Naturally, all of the functionality from the initial version of the Noun Phrase Extractor, the Parser, is inherited by the ModelMaker tool. However, because the ModelMaker tool was developed using object-oriented programming, these two components can be separated or modified at any time. All of the communication between the ModelMaker component and the Noun Phrase Extractor component is done by message passing. The Noun Phrase Extractor and ModelMaker combination works well in making the Parser useful to a modeler.

The enhanced version of the ASPIN Semantic Analyzer, the Template Constructor, is not yet integrated into the ModelMaker binary code. ModelMaker is an executable application written in Microsoft Visual C++, and the Template Constructor is a set of SWI Prolog rules that will always require an SWI Prolog interpreter. The ModelMaker tool communicates with the Template Constructor through temporary files. Using this method, the ModelMaker tool can evoke any of the functions of the Template Constructor. For each sentence ModelMaker receives from the Noun Phrase Extractor, an entry is added to the Template Constructor input file. Once all of the sentences are received, the Template Constructor control file is called from the SWI Prolog interpreter. This control file contains the controlling rules prepared by the ModelMaker component to control the behavior of the Template Constructor. Once the output file is written by the

Template Constructor, ModelMaker will read this output file and present its contents to the modeler.

ModelMaker was designed to correctly identify standard document formats of the specification document without any markup symbols. The current implementation of the ModelMaker tool is able to correctly identify a document hierarchy that uses decimal section numbering. The ModelMaker tool maintains this hierarchy information in its internal document object. This information is useful in maintaining traceability between the model and the source document.

The VHDL Identifier Extractor reads VHDL files generated by the Synopsys Graphical Environment Shell standard format. For other hardware description language, however, this version of ModelMaker may not be able to collect all of the identifiers. This is because the algorithm used in this implementation is based on the position of the identifiers relative to VHDL keywords. This limitation can be eliminated by modifying the VHDL reader algorithm of the ModelMaker tool.

The search and sort capabilities in the current ModelMaker application are useful features. Using the search ability, the modeler can retrieve the most relevant information from the source document to the particular section or part that he/she is working with. The “And” and “Or” Boolean search types are useful for multi-word keyword searches. The modeler can specify whether he/she wants to view the available information for each of the words or all of the words at any given time. Moreover, the ability to sort the search results *sequentially*, *relevantly*, *structurally*, and *behaviorally* provides useful options for the modeler. The modeler can choose whichever option is best for his/her current needs. All of these capabilities should increase the modeler’s productivity, because he/she can view the necessary information directly instead of going through all of the specification documentation.

As an overall system, ModelMaker does fulfill its objective. This ModelMaker tool shows that natural language analyzing capability can be a useful tool for hardware

designers. Not only can such a tool help the designer to understand the real category carried by the specification document, but it can also help the designer to modify his/her designs based on modified document specifications.

6.2 System Limitations

Currently, the executable version of ModelMaker can be run under the Windows 95 environment only. This is because its graphical user interface was developed under Microsoft Visual C++. The main engine behind the ModelMaker tool, however, can be rebuilt for any other platform because it was written in ANSI C++ Standard.

Another minor limitation of this ModelMaker system is the processing time. Because this is a prototype system, many of the algorithms were written for functionality instead of performance. Consequently, some of these algorithms cause a long processing time. A 486-based machine or higher should be able to run this ModelMaker application with no problem. Naturally, the higher the speed of the processor, the better it is for this ModelMaker application.

6.3 Future Enhancements

As the research continues, many enhancements can still be added to this ModelMaker tool. These enhancements can be either in the performance direction or in the functionality direction. Most of the ModelMaker code uses the object-oriented programming method. This fact should make the code modifiable and expandable.

In the performance direction, one could try to tune each of the algorithms to increase the analyzing speed. If each of the algorithms were optimized, the overall performance of the ModelMaker tool could be increased considerably. Another possible change to increase the performance would be to fully integrate the Template Constructor component inside ModelMaker. Because all of the communication would be done

internally, this integration could increase the performance substantially. This integration is in progress under a different project.

To improve functionality, one could modify ModelMaker to add more capabilities, such as a VHDL code checker and a search engine for the Template Constructor results. Having the VHDL code checker capability would allow the modeler to check his/her model code while modeling. Moreover, searchable Template Constructor results would help when the modeler wished to focus on a particular part or section of the device.

References:

[1] Alfred V. Aho and Jeffrey D. Ullman, The Theory of Parsing, Translation, and Compiling, Prentice Hall Inc., New Jersey, 1973.

[2] James R. Armstrong and F. Gail Gray, Structured Logic Design with VHDL, Prentice Hall PTR, New Jersey, 1993.

[3] Jim Armstrong, "Process-Level Modeling with VHDL," Proc. of 1998 International Verilog HDL Conference and VHDL International Users Forum, Santa Clara, California, Mar. 16-19, 1998.

[4] Grady Booch, Object-Oriented Analysis and Design with Applications (Second Edition), Addison-Wesley Publishing Company, California, 1994.

[5] David G. Burnette, A Graphical Representation for VHDL Models, Master's Thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1988.

[6] Cadence Design System Inc., Cadence Softwares, Cadence Design System Inc., http://www.cadence.com/software/software.html.

[7] W. F. Clocksin and C. S. Mellish, Programming in Prolog (Fourth Edition), Springer-Verlag, Heidelberg, 1994.

[8] Walling R. Cyre, "Mapping Design Knowledge from Multiple Representations," Proc. Of 1991 IEEE International Conference on Computer Design (ICCD'91), Boston, Massachusetts, October 1991.

[9] Walling R. Cyre, "A Requirements Sublanguage for Automated Analysis," International Journal of Intelligent Systems, 10 (7), 665-689, July 1995.

[10] Walling R. Cyre and Andreas Gunawan, "ModelMaker: A Tool for Rapid Modeling From Device Descriptions," Proc. of 1998 International Verilog HDL Conference and VHDL International Users Forum, Santa Clara, California, Mar. 16-19, 1998.

[11] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan and Jie Gong, Specification and Design of Embedded Systems, Prentice Hall PTR, New Jersey, 1994.

[12] Rob Greenwood, Semantic Analysis for System Level Design Automation, Master's Thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1992.

[13] David Harel and Amnon Naamad, The STATEMATE Semantics of Statecharts, i-Logix Inc., July 1996.

[14] David Harel and Michal Politi, Modeling Reactive Systems with Statecharts, i-Logix Inc., June 1996.

[15] Intel, Peripherals, Intel Corporation, Mt. Prospect, Illinois, 1990.

[16] Balraj Singh, John Wicks, Philip Wright and James R. Armstrong, "The Modeler's Assistant: A CAD Tool for Behavioral Model Development," Proc. of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications, Ottawa, Ontario, Canada, April 26-28, 1993.

[17] Stanley B. Lippman, Inside the C++ Object Model, Addison-Wesley Publishing Company, Massachusetts, 1996.

[18] Microsoft, Visual C++ Programmer's Guide, Microsoft Press, Washington, 1997.

[19] Balraj Singh, A Parametrized CAD Tool for VHDL Model Development with X Windows, Master's Thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1990.

[20] John F. Sowa, Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley Publishing Company, Massachusetts, 1984.

[21] Synopsys Inc., Synopsys Online Documentation v1997.08 2.1.0, Synopsys Inc, 1997.

[22] Jan Wielemaker, SWI-Prolog Information Sheet, University of Amsterdam, http://swi.psy.uva.nl/usr/jan/SWI-Prolog.html.

[23] Terry Winograd, Language as a Cognitive Process; Vol. 1: Syntax, Addison-Wesley Publishing Company, Massachusetts, 1982.

[24] Patrick Henry Winston, On to C++, Addison-Wesley Publishing Company, Massachusetts, 1994.

[25] Michael J. Young, Mastering Microsoft Visual C++ 4, Sybex Inc., California, 1996.

APPENDIX A: MODELMAKER V3.8 USER MANUAL

Introduction

This User Manual is designed to help device modelers in using the ModelMakerIII tool. This discussion gives a short description on what the ModelMakerIII tool is, and how the modeler can utilize it. Following this discussion, the system requirements of ModelMakerIII are given. A listing of the needed files to run ModelMakerIII follows these two topics. Moreover, an installation guide is also presented here. For more detailed information about any particular feature of ModelMaker, refer to the body of this thesis.

About ModelMakerIII

The ModelMakerIII is the third generation of the ModelMaker tool. It is a Windows 95-based application that facilitates device modeling by analyzing the specification document of the device and providing search facilities.

ModelMakerIII System Requirements

The minimum system requirements for the ModelMakerIII tool is a 486-based (or better) machine, with a minimum of 10 megabyte free space on the hard drive and using the Windows 95 (or higher) operating system. There is no minimum RAM (Random Access Memory). Naturally, the more RAM the system has the faster the ModelMaker tool performs its processing.

ModelMakerIII can also be invoked under the Windows NT operating system. However, it has not been thoroughly tested in the Windows NT environment.

To run the Template Constructor component of the ModelMakerIII tool, the SWI Prolog interpreter is required. The executable for this interpreter needs to be accessible from the computer where the ModelMakerIII tool is installed. This SWI-Prolog interpreter is free for non-commercial users. More information on the SWI Prolog interpreter can be found in the SWI-Prolog Information Sheet (http://swi.psy.uva.nl/usr/jan/SWI-Prolog.html).

Application Files

Listed below are the needed application files of the ModelMakerIII tool. All of these files are essential to run the ModelMakerIII tool properly. It is important that the modeler place all of these files in one directory.

- **ModelMakerIII.exe** (binary executable).
  This is the main executable file of the ModelMakerIII application. Click on the icon of this file to run ModelMakerIII.

- **Rdict.txt** (ASCII file).
  This file contains the standard dictionary for ModelMakerIII. Currently, this dictionary contains lexical knowledge of about 4000 words. This file can be edited at any time. An annotated screen capture from a small part of this dictionary can be seen in Figure A-1 to illustrate the formatting of this dictionary file.

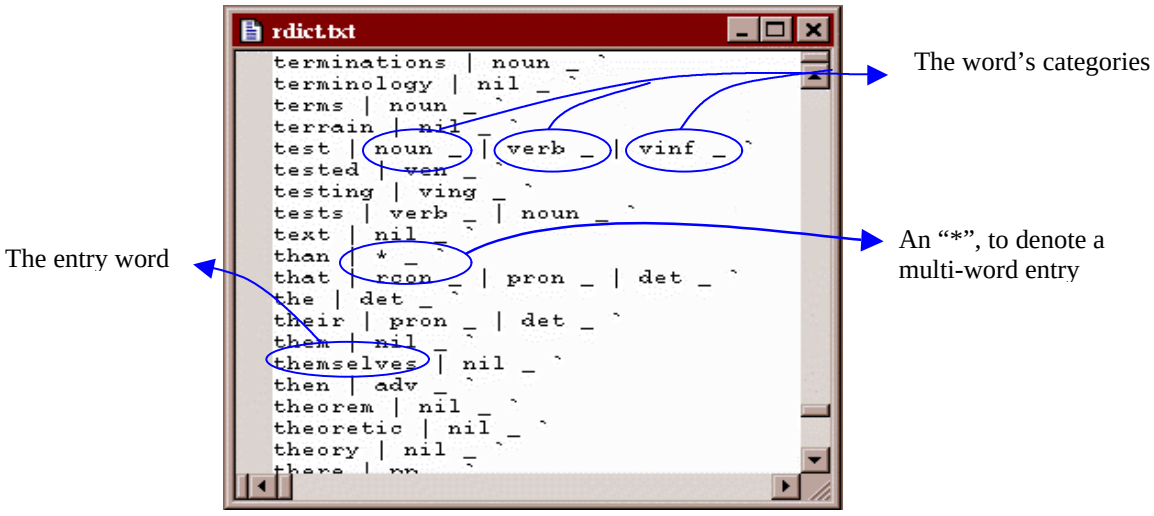


Figure A-1: Captured and Annotated Standard Dictionary File

Each line in this file represents one entry in the dictionary. Each entry consists of a vocabulary word and a list of one or more lexical categories (parts-of-speech). Each category consists of a pipe "|" followed by a category indicator followed by an underscore. A single quotation mark marks the end of an entry. These entries need to be sorted alphabetically.

An asterisk, in the category field denotes that one or more multi-word phrases that ended with the entry word are listed in the multi-word dictionary. Once the Noun Phrase Extractor finds this asterisk, it invokes the multi-word dictionary object to check for another possible category. For example, in Figure A-1, the asterisk sign for the entry word *than* refers to the multi-words *farther than*, *faster than*, *fewer than*, *greater than*, and *less than*, which are listed in the multi-word dictionary.

- **Mdict.txt** (ASCII file).

  This file contains the multi-word dictionary knowledge for ModelMakerIII. The user is encouraged to make more entries at any time. However, each time a new multi-word entry is added to this multi-word dictionary, the standard

dictionary file needs to be updated with the multi-word tag. See discussion on the "rdict.txt" file above for more information about this multi-word tag.

The entry-field structure of the multi-word dictionary file, "mdict.txt," is similar to the entry-field structure of the standard dictionary file. Each line of this multi-word dictionary file also represents one multi-word entry in the dictionary. Each of the entry lines is also divided into two or more fields as in the standard dictionary file. The only difference between the two entry field structures is that the multi-word dictionary entry field structure allows more than one word in the first field. A screen capture from a part of this dictionary can be seen in Figure A-2.

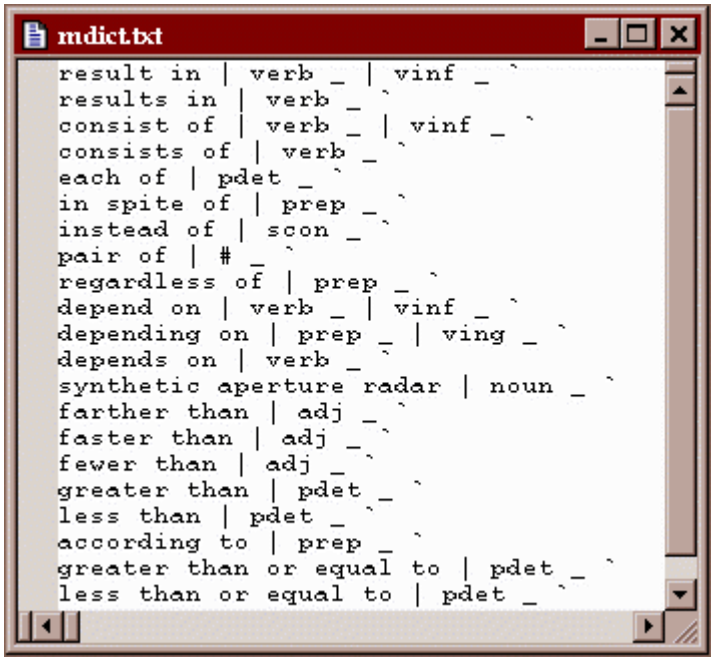


Figure A-2: Captured Multi-Word Dictionary File

- **Gram.txt** (ASCII file).

  This file contains the grammar rules used by the ModelMakerIII application. Editing this file is not recommended, as this can cause an abnormal operation of the ModelMakerIII tool. Add rules to this grammar file only after a very careful consideration. More information on this grammar file can be found in section 4.2.3.3 of the thesis. A captured image of part of the grammar file is presented in Figure A-3.

  This grammar file contains five main fields on each line. Each of the fields is separated by a space character. The first field is called the rule ID field. This field contains the grammar rule identifier for this entry. The second field is the constituent count field. This field specifies the number of constituents that are needed to create the product of this rule. The constituents are listed in the fifth and subsequent fields. They are placed last to allow a variable number of constituents. The current Noun Phrase Extractor limits this variable to the range of one to six constituents. The third field is not used by ModelMaker. The fourth field is the result field. This field stores the new category of the combined constituents.

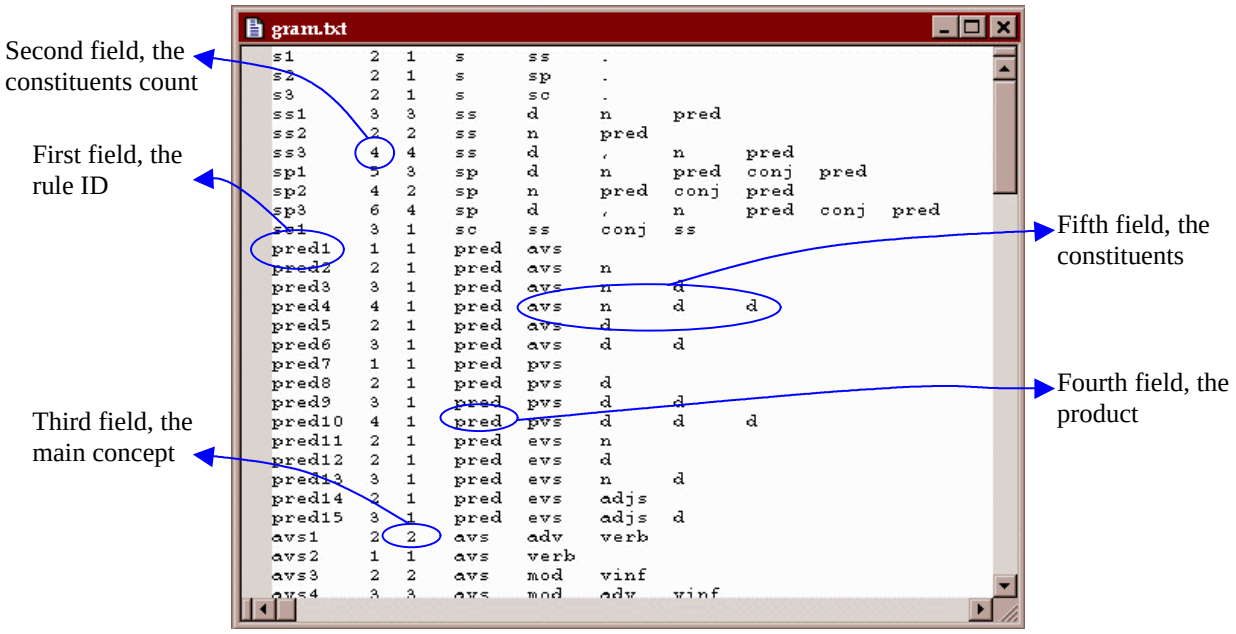


Figure A-3: Captured & Annotated Grammar File

- **Behavior.dic** (ASCII file).

  This file contains the list of the behavioral words used by the ModelMakerIII tool to rank the search results. More information on how this file is used by the ModelMakerIII tool can be found in section 4.4.5 of the body of this thesis. This dictionary file uses the same exact format as the standard dictionary file. The only difference is that the list of lexical category can be left empty ("nil") as the value will not be used by ModelMaker. To illustrate this, a behavior dictionary file is shown in Figure A-4.

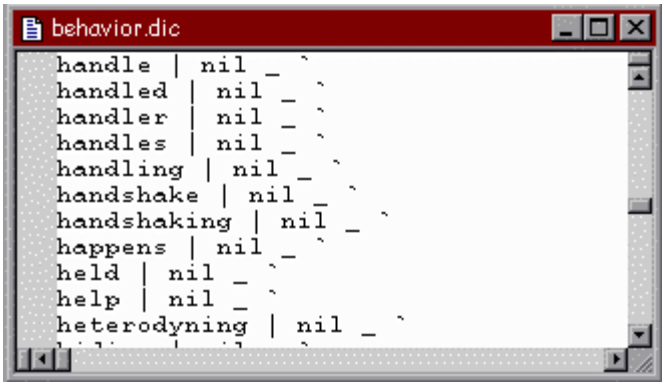


Figure A-4: Behavior Dictionary File

- **Structure.dic** (ASCII file).

  This file contains the list of the structural words used by the ModelMakerIII tool to rank the search results. Refer to section 4.4.6 of this thesis for more information on how the ModelMakerIII tool utilizes this file. Similar to the behavioral dictionary file, this dictionary file uses the same exact format as the standard dictionary file. A structural dictionary file is shown in Figure A-5.

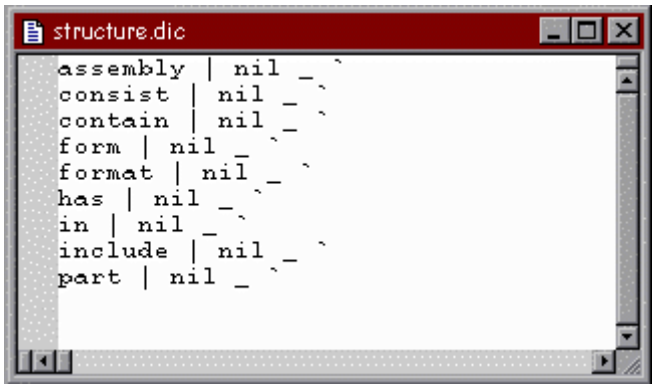


Figure A-5: Structural Dictionary File

- **Notepad+.exe** (binary executable).

  This is an ASCII viewer application (a free extension of the Windows Notepad application). This external application is required, since some commands in ModelMakerIII require an ASCII viewer. If a different ASCII viewer is preferred, simply replace this file with the new viewer's executable file. "Notepad+.exe" file name must be used for the viewer.

File "main.pl," "cannon.pl," "roots.pl," and "semrules.pl" are ASPIN components and are not distributed with ModelMakerIII.

ModelMakerIII Input Files

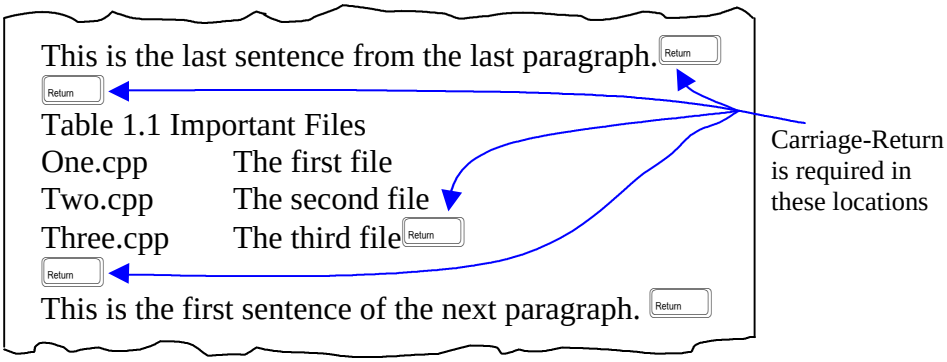
ModelMaker requires two main input files, the document-specification text file and the basic VHDL code file. The modeler should be able to obtain these two input files directly from the device documentation. Device documentation is assumed to consist of a behavioral and structural description of the device, a high-level block diagram, and possibly a pin description table and other tables.

The source document input file. A scanner with an Optical Character Recognition (OCR) program should be able to recover most of the text. This ASCII text file forms the source document file that is needed by ModelMaker. In order to store this information in a document data structure, ModelMaker attempts to recognize the format of this specification document. ModelMaker separates the document into five element types: *section header*, *paragraph*, *sentence*, *table*, and *figure*. More detailed definitions of these element types are presented in the CSpecification Class section (4.3.1.1).

In dividing the document into those five elements, ModelMaker uses the following rules:

- Elements are separated by two line-feeds or two carriage-returns.
- Each section must have a section header (this rule can be violated for the first section).
- The section header must begin on a new line and be written in the format of "*#*[*.#*] *Caption.*" In this format, "#" represents digits from zero to nine, and "Caption" can represent any string followed by two carriage-returns. The part inside the brackets can be omitted or repeated. Examples of valid section headers would include "1 Introduction", "2.1 Search Algorithm", or "2.1.3 Multiple Search".

- Tables and figures must be introduced by writing the word *Table* or *Figure* (whichever is more appropriate) as the first word. An example of a valid table element can be seen below.



- Any other element (group of sentences starting and ending with two carriage-returns) that is not recognized as a section, table or figure will be called a *paragraph*.
- A period is used to declare the end of a sentence. Each group of words that is inside a paragraph and ends with a period is called a sentence. An exception to this rule can occur with the last sentence in the paragraph. A paragraph can contain an unlimited number of sentences.

The VHDL model file. The high-level block diagram is equally important to the document-specification text file. By redrawing this diagram into a schematic-capture-application that is capable of generating a VHDL file, the modeler can use this VHDL file as the framework for his/her modeling using ModelMaker. If schematic capture is not available, the user can generate a text file containing an entity declaration with the name of the device and an empty architecture body. Then, give the file a ".vhd" extension.

Test Files. For test purposes, two sets of sample input files are included in the distribution disk. The ModelMakerIII tool allows the user to specify the names of the input files. Therefore, these files can be replaced at any time by the modeler's own input files. These sample files are listed below.

- **Dma8237a.vhd** (ASCII file).

  This file is the VHDL file generated by the Synopsys Graphical Environment shell for the structural schematic of the 8237A DMA controller.

- **Dma8237a.txt** (ASCII file).

  This file is the text file scanned from the specification document of the 8237A DMA controller.

Other Files

These are the list of the other files that may be found in the same directory with ModelMakerIII. These files are not as essential but support demonstration of some ModelMaker features.

- **Tree.txt** (ASCII file).

  This file contains preprocessed parse trees of the 8237A DMA controller document. When the user presses the "Show-Tree" button, the ModelMaker tool will search for this file, read it, and display its content in the Source Window. This capability is added to the ModelMaker tool, as the actual process of creating parse tree file can be time consuming. To re-create this file, put a check on the "Re-capture the sentence parse tree" check-box when specifying the input files (shown figure A-6 and explained in the Using ModelMakerIII section of this user manual).

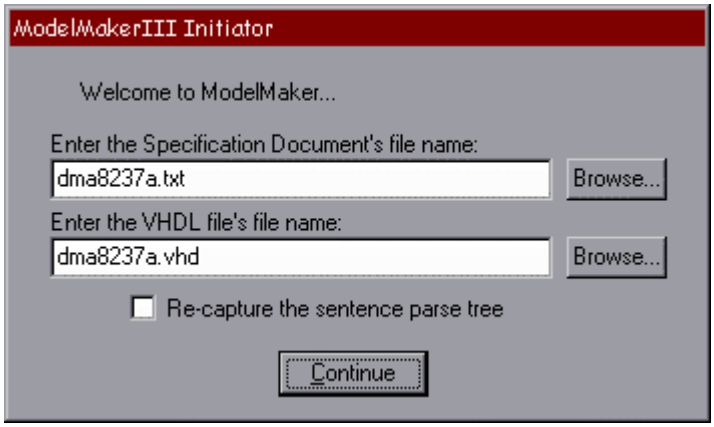


Figure A-6: The Dialog Box Used to Specify the Location of the Input Files

Once the main ModelMakerIII window is up, press the "Show-Tree" button. Be aware that this process may take a very long time. The current parse trees will be shown in the Source Window. The content of "Tree.txt" file is copied from this Source Window.

- **_Out.txt** (ASCII file).

  This file contains templates produced by the Template Constructor from the 8237A DMA controller document. When the user presses the "Template-View" button, the ModelMaker tool will search for this file and display its contents to the Source Window.

- **ModelMakerIII.log** (ASCII file).

  This file is created the first time the ModelMakerIII tool is executed in the system. This file is used by the ModelMakerIII tool to track the user configuration.

- **ReadMe.txt** (ASCII file). This text file contains information about the ModelMakerIII tool. This file is not needed directly by the ModelMakerIII tool and can be deleted at any time. This file is just an information file for the ModelMakerIII tool users.

- **Notepad+ License.txt** (ASCII file).

  This file holds the license agreement for the Notepad+ application. The ModelMakerIII tool uses this Notepad+ application as an external ASCII file viewer. This Notepad+ is used as it handles big files efficiently. This Notepad+ application can be freely distributed as long as it is accompanied by this license file.

ModelMakerIII Installation Guide

To install the ModelMakerIII tool in a system, create a new directory and copy all of the ModelMakerIII distribution files to this newly created directory. The ModelMakerIII tool is an independent application. As long as it can find its own components, it will run with no problem. Once all of these distribution files are placed in one directory, the ModelMakerIII tool is ready to be launched.

Using ModelMakerIII

This section is designed as a quick reference for the modeler in using the ModelMakerIII tool.

- **Initiate**

  To initiate the ModelMakerIII tool, simply open the ModelMakerIII directory and click on the ModelMakerIII icon (shown in Figure A-7).

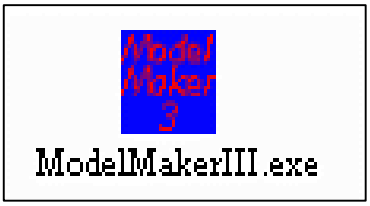


Figure A-7: ModelMakerIII Executable file

Pressing this ModelMakerIII button will bring up the input file dialog box (as shown in Figure A-8).

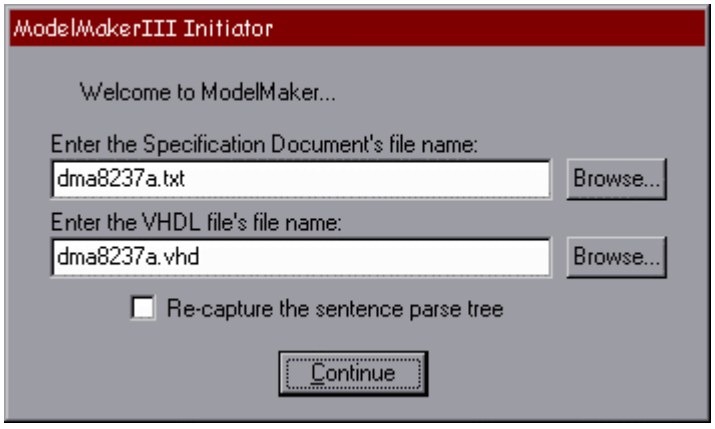


Figure A-8: The Dialog Box Used to Specify the Location of the Input Files

Specify the input files in that input file dialog box and press the "Continue" button. If this is the first time the ModelMaker tool is executed, another dialog box will pop up, querying the user about the location of the SWI-Prolog executable file. This dialog box can be seen in Figure A-9. The ModelMakerIII tool saves the user configuration inside the "ModelMakerIII.log" file. Therefore, the modeler only needs to specify this SWI-Prolog location once.

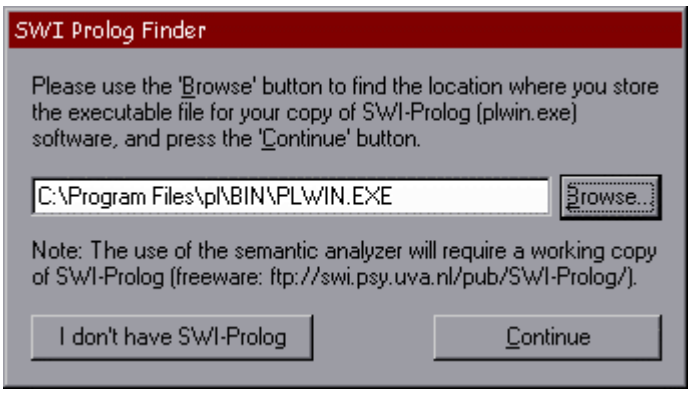


Figure A-9: The Dialog Box Used to Specify the Location of the SWI-Prolog Executable

Since the SWI Prolog is not needed with the ModelMakerIII distribution, just press the "I don't have SWI-Prolog" button. This should initiate the ModelMaker tool, and bring up the main ModelMaker window (as shown in Figure A-10).

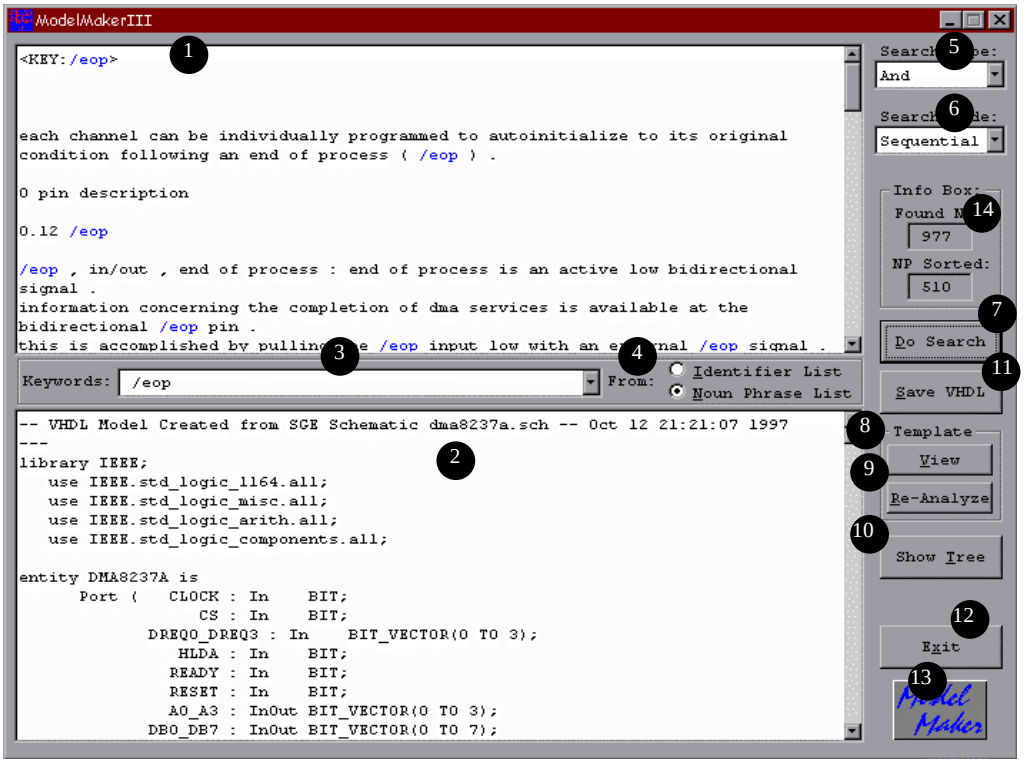


Figure A-10: The ModelMaker Tool Main Window

Depending on the size of the input documents, the initial processing may require some time. Listed below, are the components of the ModelMakerIII tool main window, as shown in Figure A-4.

1. **Source Window** - Customized Edit Window.

   The Source Window, placed in the top portion of the main working window, is where the analyzed specification document elements will be displayed by ModelMaker. This window allows the user to select and copy some or all of the text and paste it into the Model Window or any

other edit windows. This window, however, is a read only edit window. No information can be entered to or removed from the Source Window. To copy information from the Source window, the modeler need to select the portion of the text that he/she wants to copy. Once the selection is made, point the mouse pointer anywhere in the Source Window and click the right mouse button. This action will bring up a pop-up, editing menu, window that allows the modeler to copy the selected information.

2. **Model Window** - Customized Edit Window.

   The Model Window, placed in the bottom portion of the main working window, is initialized with the basic VHDL model. The Model Window is a read-write edit window. Information can be entered to and removed from the Model Window at any time. To cut, copy, or paste information from and to this Model Window, follow the same procedure with copying information from the Source Window. First, make the selection. Second, point the mouse pointer inside the window. Third, click the right mouse button to bring up the editing menu.

3. **Keywords Window** - Editable Drop-Down List.

   This drop-down list box is where the list of the identifiers or the list of the noun phrases will be presented to the modeler. The modeler can either choose one of the keywords listed there, or type his/her own phrase in the keywords window. This keyword is used by the ModelMaker tool as the search phrase.

4. **Keywords Type** - Radio Buttons.

   This radio control button lets the modeler choose whether to display the identifier keyword list or the noun phrase keyword list in the Keywords Window.

5. **Search Type** - Drop-Down List.

   This drop-down list allows the modeler to decide whether he/she wants to perform a Boolean “and” search or a Boolean “or” search. This Search Type drop-down list is only useful for the multi-word keywords.

6. **Search Mode** - Drop-Down List.

This drop-down list allows the modeler to decide whether he/she wants to sort the search results *sequentially*, *relevantly*, *behaviorally*, or *structurally*.

7. **Do Search** - Button.

   The modeler needs to press the "Do-Search" button every time he/she wants to view the information on a new keyword phrase. When this button is pressed, ModelMaker will show the returned information on the selected search key in the Source Window. To perform the search, the modeler can simply pick any of the keywords in the keywords list, set the search type and the search mode, and press the "Do Search" button. Any information that is found by the search engine will be presented to the modeler in the Source Window. The modeler can also select any of the words from the Model Window (by highlighting them) and press the "Do Search" button. When any of the words in the Model Window is highlighted, the search engine of the ModelMaker tool will use the words inside the selection as its keywords. The ModelMaker tool also allows the user to type his/her own words inside the Keywords Window. This capability allows the modeler to perform the search on any keywords.

8. **Template-View** - Button.

   The "Template-View" button deals with the external Template Constructor. By pressing this button, ModelMaker will read the pre-analyzed results of the Template-Constructor, and present it in the Source Window.

9. **Template Re-Analyze** - Button.

   The "Template-View" button deals with the external program, Template Constructor. By pressing this "Template-View" button, ModelMaker will invoke the Template Constructor, and present the output in the Source Window. If the Template Constructor is installed, this button can be used to generate the semantic analysis of a new source document. This process may require a lot of time.

10. **Show-Tree** - Button.

    ModelMaker also gives the option to the modeler to view the document parse trees. This can be done by pressing the "Show-Tree" button. This capability is used mainly for debugging purposes. By viewing the tree, one can see how the sentences are parsed within ModelMaker. As the process of constructing these trees also requires a lot of time, a copy of a preprocessed tree is stored in the ModelMaker local directory. By default, pressing this button will cause ModelMaker to read from the local file. If the modeler wanted to reconstruct the parse trees, he/she can select this option during ModelMaker initialization.

11. **Save VHDL** - Button.

    Use this button to save the VHDL model code in the Model Window back to the VHDL file. Pressing this button will overwrite the old VHDL model file.

12. **Exit** - Button.

    This button is provided to terminate the ModelMakerIII tool. The close box ("X" on the right top of the window) is disabled.

13. **ModelMaker** - Button.

    This ModelMaker button is the "about" button for the ModelMakerIII tool. By pressing this button, the modeler can view the version and copyright information about this ModelMakerIII tool.

14. **Info Box** - Static Text.

    This info box is used by the ModelMakerIII tool to present some information about the specification document back to the modeler. The information that can be found in this static text box is the total number of noun phrases found in the text and the number of distinct sorted noun phrases that constitute the noun phrase index.

Copyrights, Trademarks and Licenses

APPENDIX B: BEHAVIORAL SCHEMATIC AND ITS INITIAL VHDL MODEL

This appendix shows a behavioral model created using the Synopsys Graphical Environment (SGE) shell and the initial VHDL model generated by the SGE shell. The screen capture of the behavioral model is shown in Figure B-1.

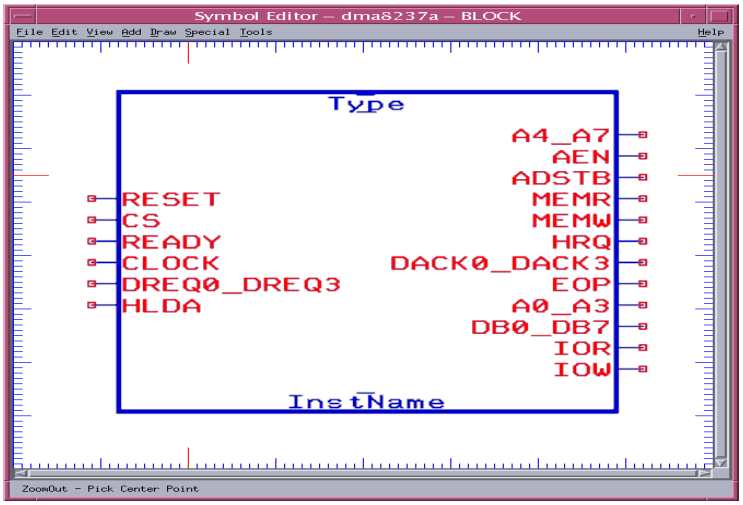


Figure B-1: The Behavioral Schematic Drawn in SGE

The initial VHDL model file created from the behavioral schematic (shown in Figure B-1) is shown in the next page.

Basic VHDL File Generated for the Behavioral Schematic

```
-- VHDL Model Created from SGE Symbol dma8237a.sym
-- Oct  5 23:27:23 1997

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_components.all;

entity DMA8237A is
   Generic ( DELAY:TIME:=0 NS );
      Port (     CLK : In    STD_LOGIC;
                  CS : In    STD_LOGIC;
               DREQ0 : In    STD_LOGIC;
               DREQ1 : In    STD_LOGIC;
               DREQ2 : In    STD_LOGIC;
               DREQ3 : In    STD_LOGIC;
                HLDA : In    STD_LOGIC := 0;
                PIN5 : In    STD_LOGIC;
               READY : In    STD_LOGIC := 0;
               RESET : In    STD_LOGIC := 0;
                 VCC : In    STD_LOGIC := +5;
                 VSS : In    STD_LOGIC := 0;
                  A0 : InOut STD_LOGIC;
                  A1 : InOut STD_LOGIC;
                  A2 : InOut STD_LOGIC;
                  A3 : InOut STD_LOGIC;
                 DB0 : InOut STD_LOGIC;
                 DB1 : InOut STD_LOGIC;
                 DB2 : InOut STD_LOGIC;
                 DB3 : InOut STD_LOGIC;
                 DB4 : InOut STD_LOGIC;
                 DB5 : InOut STD_LOGIC;
                 DB6 : InOut STD_LOGIC;
                 DB7 : InOut STD_LOGIC;
                 EOP : InOut STD_LOGIC;
                 IOR : InOut STD_LOGIC;
                 IOW : InOut STD_LOGIC;
                  A4 : Out   STD_LOGIC;
                  A5 : Out   STD_LOGIC;
                  A6 : Out   STD_LOGIC;
                  A7 : Out   STD_LOGIC;
               ADSTB : Out   STD_LOGIC;
                 AEN : Out   STD_LOGIC;
               DACK0 : Out   STD_LOGIC;
               DACK1 : Out   STD_LOGIC;
               DACK2 : Out   STD_LOGIC;
               DACK3 : Out   STD_LOGIC;
                 HRQ : Out   STD_LOGIC;
                MEMR : Out   STD_LOGIC;
                MEMW : Out   STD_LOGIC );
end DMA8237A;

architecture BEHAVIORAL of DMA8237A is
   begin

end BEHAVIORAL;

configuration CFG_DMA8237A_BEHAVIORAL of DMA8237A is
   for BEHAVIORAL

   end for;

end CFG_DMA8237A_BEHAVIORAL;
```

APPENDIX C: STRUCTURAL SCHEMATIC AND ITS INITIAL VHDL MODEL

This appendix shows a structural model created using the Synopsys Graphical Environment (SGE) shell and the initial VHDL model generated by the SGE shell. The screen capture of the structural model is shown in Figure C-1, and the initial VHDL model generated from it is shown in the next page.

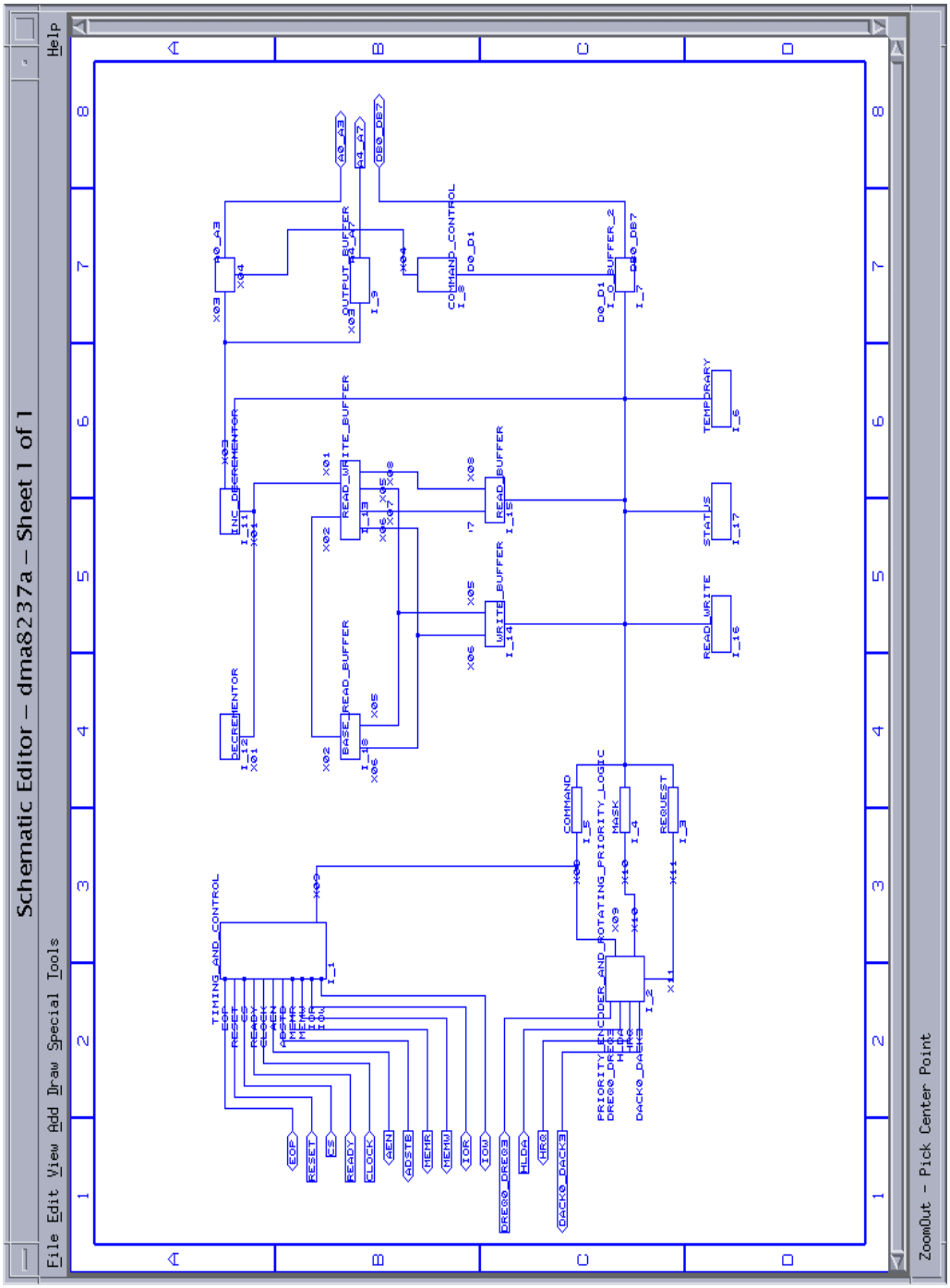


Figure C-1: The Structural Schematic Drawn in SGE

Basic VHDL File Generated for the Structural Schematic

```
-- VHDL Model Created from SGE Schematic dma8237a.sch -- Jan 30 16:27:10 1998

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_components.all;

entity DMA8237A is
      Port (   CLOCK : In    BIT;
                  CS : In    BIT;
             DREQ0_DREQ3 : In    BIT_VECTOR(0 TO 3);
                HLDA : In    BIT;
               READY : In    BIT;
               RESET : In    BIT;
               A0_A3 : InOut BIT_VECTOR(0 TO 3);
             DB0_DB7 : InOut BIT_VECTOR(0 TO 7);
                 EOP : InOut BIT;
                 IOR : InOut BIT;
                 IOW : InOut BIT;
               A4_A7 : Out   BIT_VECTOR(0 TO 3);
               ADSTB : Out   BIT;
                 AEN : Out   BIT;
             DACK0_DACK3 : Out   BIT_VECTOR(0 TO 3);
                 HRQ : Out   BIT;
                MEMR : Out   BIT;
                MEMW : Out   BIT );
end DMA8237A;

architecture SCHEMATIC of DMA8237A is
   signal      N_1 : BIT;
   signal      N_2 : BIT;
   signal      N_3 : BIT;
   signal      N_4 : BIT_VECTOR;
   signal      N_5 : BIT_VECTOR;
   signal      N_6 : BIT_VECTOR;
   signal      N_7 : BIT;
   signal      N_8 : BIT;
   signal      N_9 : BIT_VECTOR;
   signal     N_10 : BIT_VECTOR;
   signal     N_11 : BIT_VECTOR;
   signal     N_12 : BIT_VECTOR;
   signal     N_13 : BIT_VECTOR;
   signal DACK0_DACK3_DUMMY : BIT_VECTOR;
   signal      HRQ_DUMMY : BIT;
   signal     MEMW_DUMMY : BIT;
   signal     MEMR_DUMMY : BIT;
   signal    ADSTB_DUMMY : BIT;
   signal      AEN_DUMMY : BIT;

   component I_O_BUFFER_2
   Generic ( DELAY:TIME:=0 NS );
      Port (   D0_D1 : InOut BIT_VECTOR(0 TO 1);
             INTERNAL_DATA_BUS : InOut BIT_VECTOR(0 TO 15);
             DB0_DB7 : Out   BIT_VECTOR(0 TO 7) );
   end component;

   component COMMAND_CONTROL
   Generic ( DELAY:TIME:=0 NS );
      Port (   D0_D1 : InOut BIT_VECTOR(0 TO 1);
                 X04 : InOut BIT );
   end component;

   component OUTPUT_BUFFER
   Generic ( DELAY:TIME:=0 NS );
      Port (     X03 : InOut BIT;
               A4_A7 : Out   BIT_VECTOR(0 TO 3) );
   end component;
```

```
component I_O_BUFFER
Generic ( DELAY:TIME:=0 NS );
   Port (     X03 : InOut BIT;
              X04 : InOut BIT;
            A0_A3 : Out   BIT_VECTOR(0 TO 3) );
end component;

component INC_DECREMENTOR
Generic ( DELAY:TIME:=0 NS );
   Port ( INTERNAL_DATA_BUS : InOut BIT_VECTOR(0 TO 15);
              X01 : InOut BIT_VECTOR(0 TO 15);
              X03 : InOut BIT );
end component;

component DECREMENTOR
Generic ( DELAY:TIME:=0 NS );
   Port (     X01 : InOut BIT_VECTOR(0 TO 15) );
end component;

component READ_WRITE_BUFFER
Generic ( DELAY:TIME:=0 NS );
   Port (     X01 : InOut BIT_VECTOR(0 TO 15);
              X02 : InOut BIT_VECTOR(0 TO 15);
              X05 : InOut BIT_VECTOR(0 TO 15);
              X06 : InOut BIT_VECTOR(0 TO 15);
              X07 : InOut BIT_VECTOR(0 TO 15);
              X08 : InOut BIT_VECTOR(0 TO 15) );
end component;

component WRITE_BUFFER
Generic ( DELAY:TIME:=0 NS );
   Port ( INTERNAL_DATA_BUS : InOut BIT_VECTOR(0 TO 15);
              X05 : InOut BIT_VECTOR(0 TO 15);
              X06 : InOut BIT_VECTOR(0 TO 15) );
end component;

component READ_BUFFER
Generic ( DELAY:TIME:=0 NS );
   Port ( INTERNAL_DATA_BUS : InOut BIT_VECTOR(0 TO 15);
              X07 : InOut BIT_VECTOR(0 TO 15);
              X08 : InOut BIT_VECTOR(0 TO 15) );
end component;

component READ_WRITE
Generic ( DELAY:TIME:=0 NS );
   Port ( INTERNAL_DATA_BUS : InOut BIT_VECTOR(0 TO 15) );
end component;

component STATUS
Generic ( DELAY:TIME:=0 NS );
   Port ( INTERNAL_DATA_BUS : InOut BIT_VECTOR(0 TO 15) );
end component;

component BASE_READ_BUFFER
Generic ( DELAY:TIME:=0 NS );
   Port (     X02 : InOut BIT_VECTOR(0 TO 15);
              X05 : InOut BIT_VECTOR(0 TO 15);
              X06 : InOut BIT_VECTOR(0 TO 15) );
end component;

component TEMPORARY
Generic ( DELAY:TIME:=0 NS );
   Port ( INTERNAL_DATA_BUS : InOut BIT_VECTOR(0 TO 15) );
end component;

component REQUEST
Generic ( DELAY:TIME:=0 NS );
   Port ( INTERNAL_DATA_BUS : InOut BIT_VECTOR(0 TO 15);
              X11 : InOut BIT );
end component;
```

```
   component MASK
   Generic ( DELAY:TIME:=0 NS );
      Port ( INTERNAL_DATA_BUS : InOut BIT_VECTOR(0 TO 15);
                  X10 : InOut BIT );
   end component;

   component COMMAND
   Generic ( DELAY:TIME:=0 NS );
      Port ( INTERNAL_DATA_BUS : InOut BIT_VECTOR(0 TO 15);
                  X09 : InOut BIT );
   end component;

   component TIMING_AND_CONTROL
   Generic ( DELAY:TIME:=0 NS );
      Port (   ADSTB : In    BIT;
                 AEN : In    BIT;
               CLOCK : In    BIT;
                  CS : In    BIT;
                 EOP : In    BIT;
                 IOR : In    BIT;
                 IOW : In    BIT;
                MEMR : In    BIT;
                MEMW : In    BIT;
               READY : In    BIT;
               RESET : In    BIT;
                 X09 : InOut BIT );
   end component;

   component PRIORITY_ENCODER_AND_ROTATING_PRIORITY_LOGIC
   Generic ( DELAY:TIME:=0 NS );
      Port ( DACK0_DACK3 : In    BIT_VECTOR(0 TO 3);
             DREQ0_DREQ3 : In    BIT_VECTOR(0 TO 3);
                HLDA : In    BIT;
                 HRQ : In    BIT;
                 X09 : InOut BIT;
                 X10 : InOut BIT;
                 X11 : InOut BIT );
   end component;

begin

   DACK0_DACK3 <= DACK0_DACK3_DUMMY;
   HRQ <= HRQ_DUMMY;
   MEMW <= MEMW_DUMMY;
   MEMR <= MEMR_DUMMY;
   ADSTB <= ADSTB_DUMMY;
   AEN <= AEN_DUMMY;

   I_7 : I_O_BUFFER_2
   Generic Map ( DELAY=>0 NS )
      Port Map ( D0_D1=>N_9, INTERNAL_DATA_BUS=>N_4, DB0_DB7=>DB0_DB7 );
   I_8 : COMMAND_CONTROL
   Generic Map ( DELAY=>0 NS )
      Port Map ( D0_D1=>N_9, X04=>N_8 );
   I_9 : OUTPUT_BUFFER
   Generic Map ( DELAY=>0 NS )
      Port Map ( X03=>N_7, A4_A7=>A4_A7 );
   I_10 : I_O_BUFFER
   Generic Map ( DELAY=>0 NS )
      Port Map ( X03=>N_7, X04=>N_8, A0_A3=>A0_A3 );
   I_11 : INC_DECREMENTOR
   Generic Map ( DELAY=>0 NS )
      Port Map ( INTERNAL_DATA_BUS=>N_4, X01=>N_6, X03=>N_7 );
   I_12 : DECREMENTOR
   Generic Map ( DELAY=>0 NS )
      Port Map ( X01=>N_6 );
   I_13 : READ_WRITE_BUFFER
   Generic Map ( DELAY=>0 NS )
      Port Map ( X01=>N_6, X02=>N_5, X05=>N_13, X06=>N_12, X07=>N_11,
                 X08=>N_10 );
   I_14 : WRITE_BUFFER
```

```
   Generic Map ( DELAY=>0 NS )
      Port Map ( INTERNAL_DATA_BUS=>N_4, X05=>N_13, X06=>N_12 );
   I_15 : READ_BUFFER
   Generic Map ( DELAY=>0 NS )
      Port Map ( INTERNAL_DATA_BUS=>N_4, X07=>N_11, X08=>N_10 );
   I_16 : READ_WRITE
   Generic Map ( DELAY=>0 NS )
      Port Map ( INTERNAL_DATA_BUS=>N_4 );
   I_17 : STATUS
   Generic Map ( DELAY=>0 NS )
      Port Map ( INTERNAL_DATA_BUS=>N_4 );
   I_18 : BASE_READ_BUFFER
   Generic Map ( DELAY=>0 NS )
      Port Map ( X02=>N_5, X05=>N_13, X06=>N_12 );
   I_6 : TEMPORARY
   Generic Map ( DELAY=>0 NS )
      Port Map ( INTERNAL_DATA_BUS=>N_4 );
   I_3 : REQUEST
   Generic Map ( DELAY=>0 NS )
      Port Map ( INTERNAL_DATA_BUS=>N_4, X11=>N_2 );
   I_4 : MASK
   Generic Map ( DELAY=>0 NS )
      Port Map ( INTERNAL_DATA_BUS=>N_4, X10=>N_3 );
   I_5 : COMMAND
   Generic Map ( DELAY=>0 NS )
      Port Map ( INTERNAL_DATA_BUS=>N_4, X09=>N_1 );
   I_1 : TIMING_AND_CONTROL
   Generic Map ( DELAY=>0 NS )
      Port Map ( ADSTB=>ADSTB_DUMMY, AEN=>AEN_DUMMY, CLOCK=>CLOCK,
                 CS=>CS, EOP=>EOP, IOR=>IOR, IOW=>IOW, MEMR=>MEMR_DUMMY,
                 MEMW=>MEMW_DUMMY, READY=>READY, RESET=>RESET, X09=>N_1 );
   I_2 : PRIORITY_ENCODER_AND_ROTATING_PRIORITY_LOGIC
   Generic Map ( DELAY=>0 NS )
      Port Map ( DACK0_DACK3=>DACK0_DACK3_DUMMY,
                 DREQ0_DREQ3=>DREQ0_DREQ3, HLDA=>HLDA, HRQ=>HRQ_DUMMY,
                 X09=>N_1, X10=>N_3, X11=>N_2 );

end SCHEMATIC;

configuration CFG_DMA8237A_SCHEMATIC of DMA8237A is

   for SCHEMATIC
      for I_7: I_O_BUFFER_2
         use configuration WORK.CFG_I_O_BUFFER_2_BEHAVIORAL;
      end for;
      for I_8: COMMAND_CONTROL
         use configuration WORK.CFG_COMMAND_CONTROL_BEHAVIORAL;
      end for;
      for I_9: OUTPUT_BUFFER
         use configuration WORK.CFG_OUTPUT_BUFFER_BEHAVIORAL;
      end for;
      for I_10: I_O_BUFFER
         use configuration WORK.CFG_I_O_BUFFER_BEHAVIORAL;
      end for;
      for I_11: INC_DECREMENTOR
         use configuration WORK.CFG_INC_DECREMENTOR_BEHAVIORAL;
      end for;
      for I_12: DECREMENTOR
         use configuration WORK.CFG_DECREMENTOR_BEHAVIORAL;
      end for;
      for I_13: READ_WRITE_BUFFER
         use configuration WORK.CFG_READ_WRITE_BUFFER_BEHAVIORAL;
      end for;
      for I_14: WRITE_BUFFER
         use configuration WORK.CFG_WRITE_BUFFER_BEHAVIORAL;
      end for;
      for I_15: READ_BUFFER
         use configuration WORK.CFG_READ_BUFFER_BEHAVIORAL;
      end for;
      for I_16: READ_WRITE
         use configuration WORK.CFG_READ_WRITE_BEHAVIORAL;
```

```
      end for;
      for I_17: STATUS
         use configuration WORK.CFG_STATUS_BEHAVIORAL;
      end for;
      for I_18: BASE_READ_BUFFER
         use configuration WORK.CFG_BASE_READ_BUFFER_BEHAVIORAL;
      end for;
      for I_6: TEMPORARY
         use configuration WORK.CFG_TEMPORARY_BEHAVIORAL;
      end for;
      for I_3: REQUEST
         use configuration WORK.CFG_REQUEST_BEHAVIORAL;
      end for;
      for I_4: MASK
         use configuration WORK.CFG_MASK_BEHAVIORAL;
      end for;
      for I_5: COMMAND
         use configuration WORK.CFG_COMMAND_BEHAVIORAL;
      end for;
      for I_1: TIMING_AND_CONTROL
         use configuration WORK.CFG_TIMING_AND_CONTROL_BEHAVIORAL;
      end for;
      for I_2: PRIORITY_ENCODER_AND_ROTATING_PRIORITY_LOGIC
         use configuration
WORK.CFG_PRIORITY_ENCODER_AND_ROTATING_PRIORITY_LOGIC_BEHAVIORAL;
      end for;
   end for;

end CFG_DMA8237A_SCHEMATIC;
```

VITA

Andreas Indra Gunawan was born in Surabaya, Indonesia, at the end of the year 1973. He came to the United States in June 1992 to attend West Virginia University (WVU) in Morgantown, West Virginia. He took an intensive English program for six months before he finally joined WVU as a freshman in January 1993. Invited by the program, he joined the WVU Honors Program from his second semester in WVU. During his undergraduate studies, he was an active member of the WVU Electric Formula Lightning Racing Car design team. He served as the vice president of the Eta Kappa Nu electrical engineering honor society, Beta Rho chapter, from 1995 to 1996. He was also an active member of the Tau Beta Pi and the Golden Key honor society. During his free time, he was working part time as a computer consultant for the WVU. On May 1996, he graduated magna cum laude from the West Virginia University with two degrees, B.Sc. in Electrical Engineering and B.Sc. in Computer Engineering.

To continue his study, he joined the Virginia Polytechnic Institute and State University, or usually called Virginia Tech, in August 1996. Again, he joined the E.E. department with concentration in computer engineering. In the first semester, he worked as a graduate research assistant under Dr. Scott F. Midkiff to study the feasibility of a semi automatic system that gathers and presents information from the Internet, on-line services and CD-ROM databases. From the second semester until his last semester in Virginia Tech, he worked as a graduate research assistant under Dr. Walling R. Cyre. The research objective was to create a tool for rapid construction of behavioral engineering models from natural language descriptions such as written specification. He wrote his thesis based on this research work. He defended his thesis on May 20th, 1998 and received his degree, Master of Science in Electrical Engineering from Virginia Tech.

His immediate future plan is to begin his employment with Texas Instruments Inc. in Dallas, Texas, as an ASIC Software System Engineer in the semiconductor group.

Andreas Indra Gunawan