THE VAT TOOL:
AUTOMATIC TRANSFORMATION OF VHDL TO TIMED AUTOMATA

by

CARL NEHME

B.S. Computer Engineering
University of Toronto, 2002

SUBMITTED TO THE DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTERS OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2004

Signature of Author:_____
Department of Aeronautics and Astronautics
May 21, 2004

Certified by:_____
I. Kristina Lundqvist
Charles S. Draper Assistant Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by:_____
Edward M. Greitzer
H.N. Slater Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

The VAT Tool:

Automatic Transformation of VHDL to Timed Automata

by

Carl Nehme

Submitted to the Department of Aeronautics and Astronautics
on May 21, 2004 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Aeronautics and Astronautics

ABSTRACT

Embedded systems have become an integral part of the systems we use today. These types of systems are constrained by both stringent time requirements and limited resource availability. Traditionally, high-integrity embedded systems operated on well understood hardware platforms. The emergence of inexpensive FPGAs (Field Programmable Gate Arrays) and ASICs (Application Specific Integrated Circuits) as operational platforms for embedded software, has resulted in the system developer having to verify both the hardware and the software components. The stringent processes used over the system development lifecycle have to be augmented to account for this paradigm shift. One possible approach is to create a homogenous formal model that accounts for both the hardware and the software components of the system. This thesis focuses on making a contribution to the extraction of formal models from the VHDL specification of the operational platform.

The research underlying this thesis was driven by the goals of: a) augmenting the system developer's verification and validation toolbox with a powerful yet easy-to-use tool; b) developing a tool that is modular, extensible, and adaptable to changing customer requirements; c) providing a transparent transformation process, which can be leveraged by both academia and industry. The thesis discusses in detail, the design and development of the VAT tool, that transforms VHDL specifications into finite state machines. It discusses the use of model checking on the extracted formal model and presents a visualization technique that enables manual inspection of the formal model.

Thesis Supervisor: I. Kristina Lundqvist
Title: Charles S. Draper Assistant Professor of Aeronautics and Astronautics

Acknowledgements

First and foremost, I would like to express my deep gratitude for my advisor, Prof. Kristina Lundqvist, who believed in me and motivated me throughout my journey to becoming a masters student. She was not just an advisor, but also a mentor and a friend.

I would also like to thank the Aero-Astro faculty who have always been supportive and keen on helping any member of the department. I would like to thank Jayakanth Srinivasan specifically for being there when I was lost and for helping me find the way when I needed someone to lean on.

I also owe my gratitude to Lars Asplund, Gustaf Naeser and Johan Furunäs for helping me see beyond the near horizon.

Thanks are also due to my parents, my siblings and my friends for supporting me and pushing me to accomplish the best.

Last but not least, I would like to acknowledge the great working environment provided by my teammates, Pee Seeumpornroj, Sébastien Gorelov, Wayland Ni, Gaston Fiore, Anna Silbovitz, and Kathryn Fischer, that made this an enjoyable experience.

Table of Contents

1. Introduction

Software and hardware (SW/HW) have manifested themselves as essential components in a lot of the systems we use today. Taking the aeronautical domain as an example, in conjunction with all the other factors affecting the safety of air travel, such as pilot competency and air traffic control efficiency, the reliability of air transportation services are highly dependent on the constituent embedded systems. The Boeing 777 for example, has 1,280 embedded processors [AIA04]. An embedded system, or a real-time system, is a software/hardware (SW/HW) system whose prime function is not that of information processing, but which nevertheless requires information processing to carry out its prime function [BW01]. The design of an embedded system is primarily influenced by the environment in which it operates.Additionally, embedded systems must satisfy stringent time requirements and hence real time design is important. These types of systems are inherently existent in a multitude of systems, and methods for verifying safety critical systems can be used to increase the confidence we have in their performance.

Till date, there have been a number of accidents where SW/HW errors such as failure to display information, incorrect application of control mechanisms etc, have been listed among the causes of the accidents. Even though in certain cases the system controllers could have averted the catastrophe, there have been many incidents where disaster was not just forthcoming, but inevitable [ICM93]. Some air accidents whose investigations listed SW/HW errors as a contributing factor are listed below.

- The Dallas/Fort Worth air-traffic system began spitting out gibberish in the fall of 1989 and controllers had to track planes on paper. [Elm90]

- A China Airlines Airbus Industrie A300 crashed on April 26, 1994 killing 264 people. Recommendations include software modifications. [SL96]

Since embedded systems form an essential part of the many systems we rely on today, it is essential, specifically in the safety critical domain, to increase the confidence we have in the safe functioning of these embedded systems.

Current embedded systems are implemented using a combination of hardware and software, and are a magnitude larger and more complex than they have been. New advances in areas such as, e.g., field programmable gate arrays (FPGAs) with embedded CPUs, predictable programming languages, and verification tools, now allows for the design of high integrity embedded systems where the boundary between hardware and software can be chosen to optimize the advantages of respective areas environments. To be able to use the embedded system in high integrity environments, stringent development procedures needs to be followed both for the hardware and software development. This thesis presents the motivation for, and specific implementation details of the VHDL to automata transformation (VAT) tool. The VAT tool transforms VHDL hardware descriptions into a formal notation amenable to verification using model checking tools.

1.1 Formal Methods

Formal methods have been used successfully in the design and analysis of both hardware and software systems. They cover a broad spectrum of mathematically based approaches that can be partitioned into:

- Tools, techniques and languages to model systems at multiple levels of abstraction, and at multiple stages of the system development lifecycle
- Analysis techniques that can be applied on the created models to prove specific system properties.

Formal methods have been advocated as a way to increase the reliability of safety critical systems, but the lack of acceptance within industry has been attributed to lack of tool support and perceived mathematical difficulty. The last decade of research in formal methods has been directed at creating stable, well understood tools that abstract the perceived mathematical difficulty through the use of either programming language-like syntax or visual representation, or a combination of both approaches. This partitioning of the formal methods space is not always visible in existing tools, but is critical from the perspective of this thesis. This thesis attempts to make a contribution to the model creation domain, which has been cited as another barrier to the adoption of formal methods [SG02].

The formal analysis techniques used both in industry and academia today are model checking and theorem proving. Model checking allows for exploration of a system's finite state space, and in doing so inconsistencies with the expected behavior can be found. The expected behavior of the system can be specified as safety properties such as invariants, deadlocks and reachability, and liveness properties such as fairness and response. Depending on the model creation process, model checking can also be used to validate timing constraints on the system. Theorem proving is used to show logical correctness, and proven absence of faults in designs. Theorem provers however require human guidance to direct the proof generation process.

This thesis presents a tool for automated formal model extraction for the VHDL specification language [1076]. VHDL is commonly used for describing hardware systems for implementation on an FPGA. Compilation of the VHDL specification results in an output that can be mapped to the actual FPGA architecture.

1.2 Technology

Traditionally, hardware systems were designed and then implemented by hardwiring the design on a chip. An application-specific integrated circuit (ASIC) was used to accomplish that task. An ASIC is an integrated circuit (IC), which is designed for specific

functionality. An ASIC therefore does not serve for general purpose use, and cannot be modified after implementation on the circuit. In the late 1980s, the availability of silicon compilers that could accept hardware description language (HDL) descriptions brought FPGAs into the forefront. Unlike an ASIC, an FPGA has the ability to be reprogrammed, and this is leading to the idea of reconfigurable computing or reconfigurable systems. HDLs such as Verilog [1364] or VHDL are used to compile the logic (system behavior) into a gate level implementation on an FPGA.

As FPGAs grow larger, faster and more capable, ASICs have started to become a less attractive solution. Also, with the reduction in prices of FPGAs, their use in the embedded systems domain has increased. Due to this ever-increasing use of FPGAs, VHDL has taken a greater role in specifying systems in the embedded systems domain [Sha86], and the VAT tool can facilitate extraction of a formal model therefore enabling formal verification.

1.3 Thesis Outline

The thesis follows the following format. In the section 2, fundamental concepts and related work is presented. Section 3 presents a detailed description of the VAT tool, while section 4 is dedicated to the description of a graphing optimizer used to create the output visualization. A number of case studies are then presented in section 5. Finally, section, 6, contains the conclusions and some reflections on possible future work. Appendix A contains the definition files for the VAT code. Appendix B then follows with a listing of the VAT tool's code.

2. Background

This chapter covers background material that relates to the VAT tool and defines important terms and concepts.

2.1 Embedded Systems

An embedded system is a specialized real-time system which is designed to accomplish a specific task and runs as a subset of a larger system. Its interface with its surrounding environment is also well defined [BW01]. An embedded system is typically required to meet important additional requirements, such as realtime constraints, that a general-purpose personal computer is not concerned with. Realtime constraints are limitations placed on the software or hardware because it nows oprates in an environment which is a function of time. Therefore, the systems must accept inputs and produce outputs following a strict schedule in order for the systems to operate properly. Usually there is no disk drive, keyboard or screen. Examples of embedded systems are chips that monitor car functions, environmental systems, security systems, and entertainment systems.

The design of embedded systems is primarily influenced by the environment in which they will eventually operate. Important issues that arise in designing embedded systems are proving that properties such as absence of deadlock, and compliance with the real-time constraints are indeed satisfied.

In designing a tool, the VAT tool, that allows us to model check VHDL specifications, we are benefiting from a formal verification technique that can show

satisfaction of real-time constraints and absence of deadlock for a specification language commonly used in the embedded systems domain.

2.2 Gurkh

The overall research goal in Gurkh is to build a framework for design, verification and execution of safety critical applications. The framework consists of both software tools for application verification and hardware platforms for execution and real-time monitoring. The Gurkh Framework is shown in Figure 2.1.

A hardware Ravenscar compliant Kernel, RavenHaRT [Sil04] (1)[1], is central to this framework and provides an environment on which to run the code. Ravenscar was defined in the 8th International Real-Time Ada Workshop as a restricted tasking profile to be used in high integrity efficient real time systems [BV97]. The Ravenscar Profile is a subset of Ada95's tasking features. The reason for these limits is to provide a tasking model that can offer determinism, be analyzed using schedulability analysis methods and also offer memory-boundedness [DB98].

In this framework, the VAT tool and some other complementary tools automatically translate (2) already existing applications implemented using a mix of Ravenscar-compliant Ada 95 code and VHDL code into an internal FSM format. This internal format can in conjunction with a formal model of a run-time kernel (RTK), be converted into specific verification languages used by different verification tools such as e.g., Uppaal [LPY97] and Kronos [DOT+], for formal verification. Other parts of the Gurkh project cover hardware implementation of the verified RTK, the RavenHaRT kernel [Sil04], and a monitoring chip for real-time monitoring of code executing on RavenHaRT. The monitoring chip (3) compares runtime attributes of the software application to the formal FSM representation of the application and RavenHaRT.

[1] The numbers in parentheses are used so that the reader may more easily follow references to Figure 2.1

Thereby, we have an environment where we can model the code and verify it, and a a monitoring chip to insure that the processes on the PowerPC and RavenHaRT execute properly.

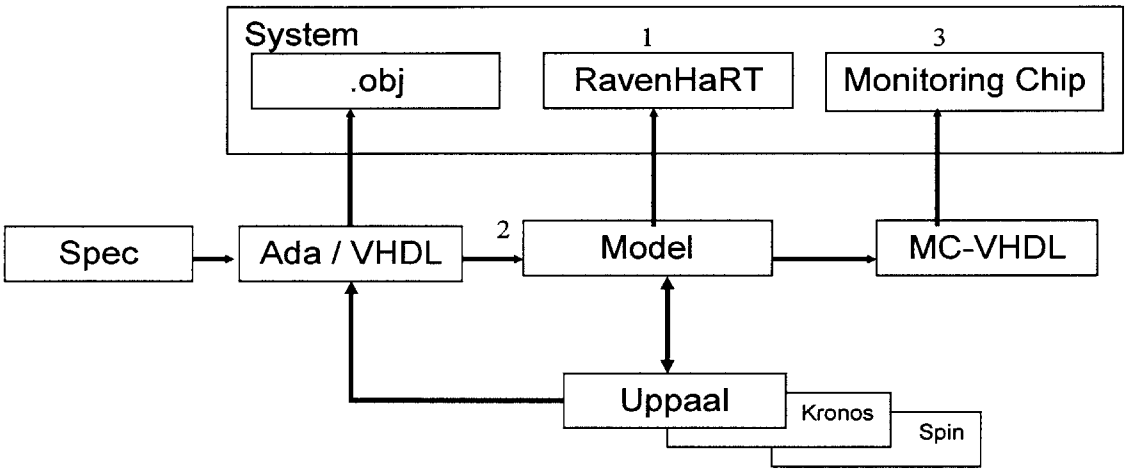


Figure 2.1. The Gurkh Framework

This thesis discusses within the context of the Gurkh project, the development of the VAT tool to translate safety critical VHDL code into a formal representation. Model checking can then be applied on this representation in order to verify kinds of properties such as liveness and deadlock and to validate that the timing constraints of the original system hold.

2.3 Verification and Validation

V&V is an acronym for validation and verification [ABC82, CDH+00]. Traditionally, engineering has relied on mathematical calculations in proving feasibility and functionality. Whereas mathematics lends itself naturally to engineering, it is not as obvious when it come to SW/HW systems.

Formal verification refers to the process of applying formal methods on formal models. Formal verification has faced a hard time in terms of being accepted by the

software public; such as software developers, software designers, and program managers. The main problem was the fact that unlike other verification techniques, formal verification required some pre-acquired education about the notations used in the formal models and the way they work in order to comprehend the results of applying the technique. Model checking and theorem proving are the two main methods of formal verification.

In the model checking approach [ACD90], a high-level model of the system under design is described in a formal notation, and the verifier automatically checks whether the given model satisfies correctness properties. Correctness properties are assertions that can cover issues such as absence of deadlock or reachability of essential states. Unlike simulation, the model checking is performed by exploring all possible interactions of the concurrent components. If an assertion is violated, the model-checker reports a counter-example as evidence of the violation. There are different algorithms used to traverse the state model defined by the system and to check if the assertions hold. Model checking also provides counter examples to assertions that do not hold.

Theorem Proving [AW84,Sch01], on the other hand, deals with the generation of a proof that shows that some statement is a logical consequence of a set of statements. The proofs prove the correctness of a program by generating a formal proof that the assertions we want the program to adhere to are logical consequences of the program itself. There is a formal language in which the conjecture, hypotheses, and axioms are written. Once everything is represented in the formal language, an attempt on constructing a proof proceeds.

The problem is not the use of formal methods on formal models (i.e. formal verification), but the generation of the formal models. The process of creating a formal model for SW/HW systems has been done traditionally using manual inspection. This process however is prone to human error and is not repeatable. There is also a problem of scoping as systems grow in size, where manual generation of formal models becomes extremely difficult, if not infeasible. It is therefore useful to have an automated tool that will extract a formal model from the systems we wish to formally verify.

Formal modeling refers to the process of generating a formal representation for some system. That representation is specified in a language used for formal modeling, such as FSMs and Statecharts [Har87]. Formal modeling can use various mathematical representations to represent entities such as code, requirement specifications, test cases, or design. Formal models can therefore be used to represent the behavior of a system in a formal manner, such as predicate calculus, which further allows us to mathematically verify using logical inference whether or not that formal model of the system functionality matched the original design requirements.

There have been many attempts at designing tools that transform specifications into formal models. Java PathFinder [HP99] and JCAT [DI99] for example translate JAVA source code to notations accepted by the Spin model checker. There were several other translators designed for other input languages. A problem with these initial attempts is that they suffered from two notable flaws. They were monolithic, in the sense that the translator and the model checker were built into one common system, hence causing modifications to either the translator or the model checker to be much too complex. As well, the translators generated a notation specific to one model checker and hence did not allow the eventual users to take advantage from the many model checkers out there.

It is a model builder that we sought to develop, whilst making sure to avoid the notable flaws just mentioned. It is also important to note that the accuracy of the transformation is the essential quality that is sought after in designing the model builders.

There are different representations that are used in generating formal models, examples are: Software Cost Reduction (SCR) [HKL97], Statecharts [Har87] and RSML [LHR99]. Finite state machines is one such representation and is the one used by the VAT tool.

2.3.1. Finite State Machines

Finite State Machines (FSMs) are mathematical models that serve as approximations of physical or abstract phenomena. An FSM is defined as a 5-tuple with states, alphabet, transition function, start state and an output assignment function [HU79]. For the purpose

of this thesis, the terms finite state machine and automaton will be used interchangeably. The difference between the two is that whilst automata contain accept states, finite state machines do not particularly contain such accept states and generally have output assignment functions for the output variables in the system.

A deterministic finite automaton is a 5-tuple: (S, Σ, T, s, A)

- a finite set called the alphabet (Σ)
- a finite set of states (S)
- a transition function ($T : S \times \Sigma \rightarrow S$).
- a start state ($s \in S$)
- a set of accept states ($A \subseteq S$)

An example of an FSM is shown in Figure 2.2. In this example, the FSM stays in the Landing_Gear_Retracted state until a certain input (Switch) causes it to go to the Landing_Gear_Released state. In the Landing_Gear_Released state, the FSM can either go back to the Landing_Gear_Retracted state or it can go into the Landing_Gear_Maintenance_State. The system cannot go into the Landing_Gear_Maintenance_State from the Landing_Gear_Retracted state for obvious reasons.

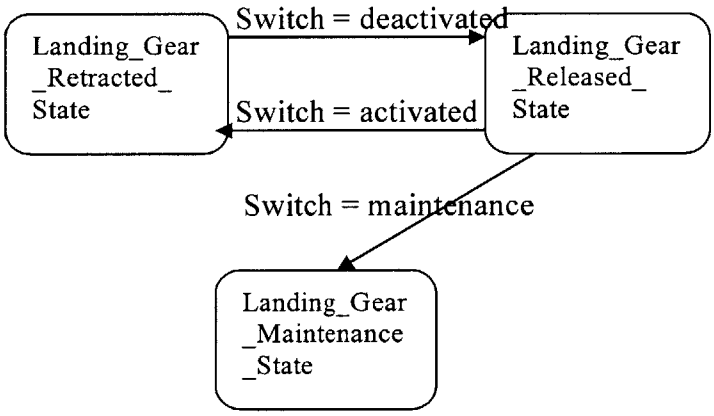


Figure 2.2. Example of an FSM for a Landing Gear

Using the above definitions, the automaton in Figure 2.2 would be defined as a deterministic automaton with the following characteristics:

- Σ = {activated, deactivated, maintenance}
- S = {Landing_Gear_Retracted_State, Landing_Gear_Released_State, Landing_Gear_Maintenance_State }
- T: Landing_ Gear_Retracted_State x deactivated → Landing_ Gear_Released_State, Landing_ Gear_Released_State x activated → Landing_ Gear_Retracted_State, Landing_ Gear_Released_State x maintenance → Landing_ Gear_Maintenance_State
- s = Landing_Gear_Released_State
- A = ∅

The VAT tool transforms code into an FSM by extracting the information that pertains to states and transitions between states. This formal representation can be used by existing tools, such as the real-time model checker Uppaal [LPY97], to verify the absence of deadlocks and other runtime properties such as state reachability.

For example, in a plane (Figure 2.2), the Landing_Gear_Retracted state should not be reached whilst the plane is on the runway. Deadlock on the other hand is the failure or inability to proceed due to two or more programs waiting on the same shared resource. Therefore the system will go nowhere and we have stagnancy.

2.3.2. Uppaal

Uppaal is a model checking tool developed as a joint research effort, between Aalborg University in Denmark, and the department of Computer Systems (DoCS) at Uppsala University in Sweden [Upp04,LPY97]. It is an integrated tool environment for the modeling, simulation and verification of real-time systems.

Uppaal consists of three main parts: a description language, a simulator and a model checker [PL00]. A guarded command language with real-valued clock variables and data types serves as the description language. The language is used to describe the system behavior as networks of automata extended with clock and data variables. The simulator provides for early detection of the possible execution paths and hence serves as a validation tool. The model checker is used to check invariant and bounded-liveness properties by exploring the symbolic state-space of a system.

A new version of Uppaal, Uppaal2k [PL00], was released in 1999 with modifications on the first release (1995). The new version, consisted of three major improvements:

- Modification to allow for easier maintenance.

- An improved graphical interface.

- Better portability to different operating systems.

The modeling language used in Uppaal2k is also richer than the one used in its predecessor. An example of the changes made to the modeling language is added support for process templates and more complex data structures, such as data variables, constants, arrays etc.

Uppaal2k was used to display the automata after they were extracted from the code by the VAT tool. Model checking was then performed on some of the models using some basic assertions (section 5.2).

2.4 VHDL

VHDL (Very High Speed Integrated Circuit Hardware Description Language) is an IEEE standard hardware description language (HDL) [BV99]. VHDL was developed by the Department of Defense in the 1980's to support the documentation of electronic systems. The air-force originally funded the development of VHDL as a means of capturing design information.

VHDL was originally intended to serve two main purposes.

- It was first used as a documentation language for describing the structure of complex digital circuits. As an official IEEE standard [1076], VHDL provided a common way of documenting circuits designed by numerous designers.
- The other purpose was providing features for modeling the behavior of a digital circuit. This enabled specifying systems in VHDL that where then used to model circuit operation.

In addition to its use for documentation and simulation, VHDL has also become popular for use in design entry in CAD systems. CAD tools are used to synthesize the VHDL code into a hardware implementation of the described circuit.

2.5 Related Work

The VAT tool has two main components:

- the part which accomplishes the state model extraction, and
- the graphing optimizer

There has been significant related work in both the model extraction and in the visualization field. Four specific tools covering model extraction are discussed in section 2.5.1: Bandera, VSPEC, Veritech, and Prevail. In section 2.5.2, the related work done in optimizing graph topologies (by optimizing node placement) to improve visualization is presented. The section mainly discusses the different algorithms that were studied before choosing the Spring Embedder algorithm used for the VAT tool.

2.5.1. Tools for Code Transformation

There have been several tools designed for translation in the context of software verification. An example is the Bandera [CDH+00] tool set, which translates Java source code to the model checking notation Promela. Promela is the input notation used by the model checker Spin [DIS99]. Bandera uses slicing to reduce the state space. Slicing involves removing sections of the code that will not affect the assertions that are to be verified. Slicing is done before the extraction of the formal model, and hence reduces the size of the eventual model. Bandera also maps the verifier's counter-example outputs back to the original source code which helps in debugging.

There has also been a lot of work on extending the capabilities of VHDL. Most of these efforts were motivated by a need to better support system design [MP98]. This has inspired some research into the formal verification of VHDL. The concept of formal

verification includes many facets and one such facet is showing conformance between a specification and an implementation. In [NIC94] a system PREVAIL is described which verifies whether or not a given implementation is functionally equivalent to its specification. This is done by converting the specification and the implementation into recursive functional forms, and the proof for their equivalence is realized by means of the Boyer-Moore theorem prover [BM88]. The difference between PREVAIL and the VAT tool is that while PREVAIL compares the implementation to the system specification, the VAT tool uses the implementation and generates from it a formal model on which model checking can be applied. Also, while PREVAIL uses theorem proving as the underlying formal verification technique, the VAT tool resorts to model checking.

VSPEC [VSP], a Larch interface language for VHDL, allows a designer to specify non-functional performance constraints. It was created in the quest for increasing support for formal methods. It allows a designer to declaratively describe the data transformation a digital system should perform and performance constraints the system must meet [MP98].

Veritech [SG02] is another related tool, but one which accomplishes a different task then a formal model translation. Veritech allows the translation between different model checking notations through an intermediate notation. The intermediate notation is CDL, core design language, in which modules can be combined in synchronous, asynchronous or partially synchronous manners, and each module is a set of first-order transitions. At present, Veritech includes translations between the notations of different formal languages such as SMV [BCM+92], Spin [DIS99], STeP [NBC+95], Petri-nets [RW98] and the core design language (CDL) [SG02]. The reason Veritech was interesting was because of the effort that has been made in order to be able to go between different model checking notations. This only stresses the importance of being able to use different model checkers to model check the same entities. This further motivated one of the goals of the VAT tool which was to be able to achieve the one to many relationship between a specification and the multiple model checking notations out there.

2.5.2. Graphing Algorithms

There have been many attempts at optimizing graph layouts to improve visualization. Most of these efforts concentrated on reducing the number of crossing edges which is an important graph quality when it comes to visual interpretation.

An interesting class of graph layout algorithms is the "force-directed placement" algorithms and specifically the Spring Embedder Algorithm. This attempts to solve the problem of node placement in a graph by simulating nodes that repel themselves while being attracted by edges in between the nodes. The benefit is that interlinked nodes appear close to each other while disconnected nodes appear distant.

A modified force-directed method, Hall's method [Hal70], resorts to matrices in representing the graph layout. The algorithm defines the laplacian of the graph as an $n \times n$ matrix L which contains the weights of the edges and the fan in and fan out of each vertex. The optimal layout of Hall's method satisfies certain constraints on the eigenvectors of the laplacian. This algorithm has better running times than the Spring Embedder Algorithm, but requires extensive memory space to store the matrices and requires matrix solutions to be reachable.

Another class of layout algorithms is the Ace graph drawing algorithm [KCH02]. It is similar to Hall's method and again uses the laplacian of matrices to find the optimal graph. It, however, uses interpolation to increase the number of nodes in the graph which eases the finding of the laplacian and hence reduces the overall computational time required.

The spring embedder algorithm [FR91] first proposed by Eades [Ead84], is a force directed layout algorithm and is a visualization algorithm that improves the aesthetic comprehensibility of a graph by minimizing the number of crossing edges. The idea of the algorithm is the also one of simulating a system of mass particles. The algorithm uses the same modeling technique used by force-directed algorithms (vertices simulate mass points repelling each other and the edges simulate springs with attracting forces) and attempts to minimize the energy of the system. In finding the configuration with

minimum energy, one can achieve a configuration with a minimum number of crossing edges due to the correlation between the two (Figure 2.3).

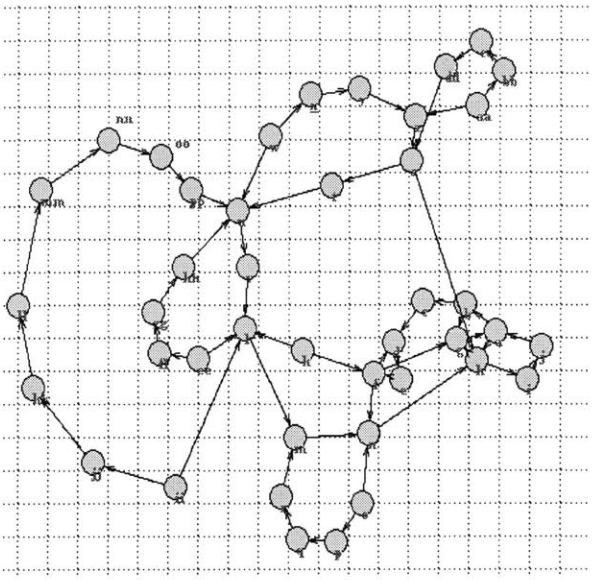


Figure 2.3. Minimization of crossing edges

For the VAT tool, a modified version of the spring embedder algorithm was designed to improve the display of the finite state machines. The modified version is suboptimal in that it does not always produce the optimal topology (minimum crossing edges) assuming a planar graph, but is less computationally expensive than the spring embedder algorithm and produces a graph that is not far off from the optimal one. The design of the modified spring embedder algorithm is discussed in detail in section 4.

3. VHDL to Automata Tool (VAT)

This chapter will discuss the design and implementation of the VAT tool and the limitations faced in the process of doing so. An overview of the tool functionality will be given, followed by a description of the tool implementation in detail.

The approach to writing VHDL code typically consists of three steps: drawing an FSM that describes the states the system can be in and the transitions between them, translating the graphical representation into a truth table, and finally generating VHDL code from the truth tables [BV99]. The tool presented here, reverses the process, starting with the VHDL code and generating the FSMs from it.

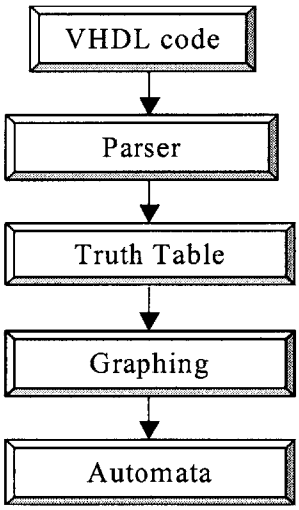


Figure 3.1. Transformation of VHDL to Automata

The tool's architecture is one of a pipe and filter architectural style, were the different stages represent the different evolutions of the code as it goes from raw VHDL to FSMs. First, the code is parsed and the syntax related issues are handled. Then, the information relevant to FSMs is extracted by analyzing the code and by using heuristics

about how FSMs are coded in VHDL. This extracted information is then stored in a truth table representation. The information in the truth table is next fed to a graphing optimizer. The function of the graphing optimizer is to find an optimal node placement such that there is minimal edge crossings which allows us to enhance readability (the graphing optimizer will be discussed separately in section 4). The state information with the corresponding graphical topology is then fed to a translator to be converted to a formal notation appropriate to the target verifier (model checker). Currently a translator for the Uppaal model checker has been constructed. However, translators for any other model checkers can be constructed too, achieving the goal of being able to verify in more than one model checker. Figure 3.1 summarizes the above process. The tool developed is a set of Packages written in Ada [Ada95].

Note that the finite state machine abstraction generated by the tool is not equivalent to the formal specification from which the code was designed. The finite state machines are generated by analyzing the VHDL code whereas the original system specifications are used as an underlying criteria in creating the code. Hence the finite state machines generated need not be equivalent to the original system specifications, especially if the VHDL code itself is not equivalent to the original systems specifications. The different packages in the tool and the functionality and data structures they represent are discussed in section 3.1. Section 3.2 provides a detailed description of how the heuristics are used to extract the formal model from the different statements present in the input VHDL specification.

3.1 Packages

The main functionality in the tool is divided into packages. The architecture is shown in Figure 3.2.

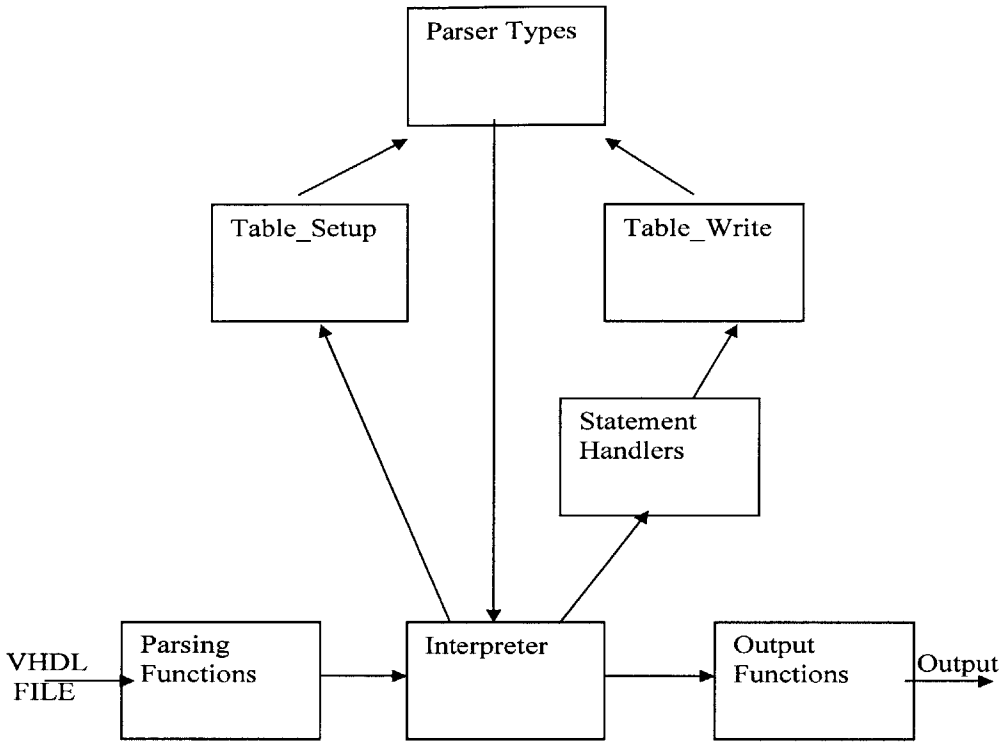


Figure 3.2. VAT Packages

The *Interpreter* package is the core of the tool, and is the main procedure that makes calls to subroutines in the other packages. The *Statement_Handlers* package contains heuristics about how FSMs are normally described, and these heuristics are used to extract the FSM models out of the code.

Parser_Types, is the package that contains all the abstract data types including the truth tables (note, there is one truth table for every state variable). The interpreter uses the *Parsing_Functions* package to parse the VHDL file and extract the syntax, and then makes calls to the *Table_Setup* package which in turn records the information by making calls to *Parser_Types*.

After the initial setup phase, as statements are encountered, the *Statement_Handlers* package is called which uses heuristics to process the statements and extract the semantics and then records those in the truth tables by accessing the *Table_Write* package. The heuristics are:

- Any signal variable that takes an enumerated set of discrete values, is a state variable, and the set of discrete values are the states that the state variable can be in.
- The set of statements that describe the conditions for the signal variable to change values from "A" to "B", represent the guard on the transition in the state diagram between state "A" and state "B".

This list of heuristics was a lot longer when the tool was first being developed. The heuristics in turn place assumptions on the way the state machines must be coded in VHDL. The greater the list of assumptions that are used, the less general the tool will be. Therefore, it was an objective from the outset to reduce the number of heuristics used, and hence the number of assumptions. The final list of heuristics boiled down to the two statements above. The assumptions that these heuristics translate into are not restrictive on the type of VHDL code that can be handled. The reason is that state machines can only be coded by having a state variable with an enumerated set of values, and hence this heuristic is not putting any assumptions on the way the state machines are coded. The same applies for the heuristic defining the transitions' mapping.

Because of the way the state variables are defined (ie. to be any variable that takes an enumerated set of values), it will be the case that the tool will generate some extra state machines which are not of interest to the user. However, the state machines are each described separately and hence the end user can pick and choose relevant state machines without being encumbered by the unnecessary ones.

Finally, when the interpreter is done parsing the file and extracting the formal model, it makes a call to the output_functions package which produces a record of the formal model. That record is used by the graphing optimizer later in the pipeline.

The packages will now each be separately discussed:

- ***Output_Functions***: This package contains functions used for printing out things to the console. Debugging information is printed out as well as a summary of the state

machines contained. There are individual functions for printing the different signals, input, and output variables found in the VHDL code. Functions also exist for printing a summary of the formal model extracted.

- ***Parser_Types:*** This package contains all the definitions for the types used to store information extracted in the parsing stage. Examples are the truth tables and the linked lists used to store all the input variables, signal variables, and output variables found while parsing the VHDL file.

Three linked lists within *Parser_Types* are used to store the inputs, signals, and output variables found (see Figure 3.3). Each node, contains the name of the variable (input/signal/output) and a pointer to the next variable in the linked list. Using a linked list allows the use of as much memory space as there are variables. It comes at the cost of being costly to traverse, but the advantages of preserving memory are worthwhile.

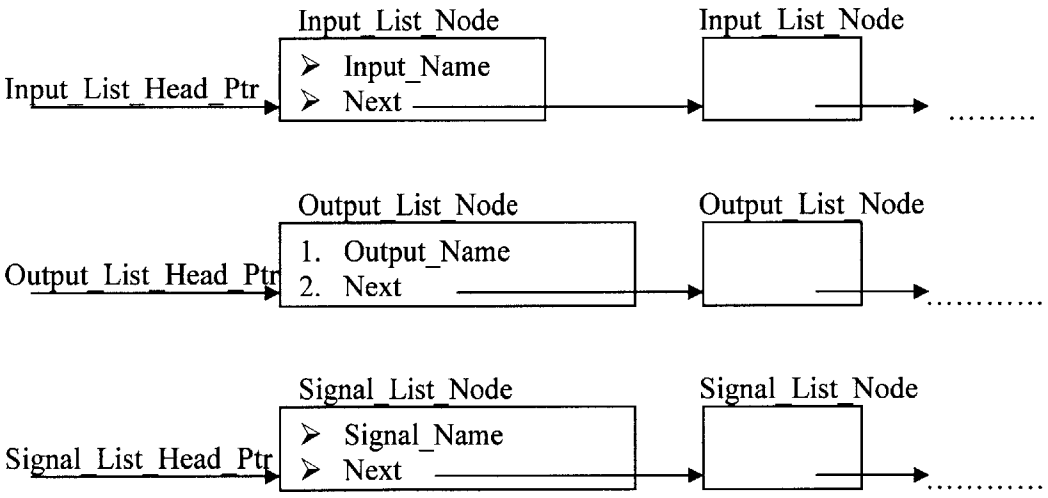


Figure 3.3. Input/Output/Signal Lists

There is also a linked list of rows, representing the rows of a truth table. A truth table consists of a set of these rows, one for each state (Figure 3.4). Each row in turn has the Present_State name, a pointer to the Next_State table (the table which based on the value

of the inputs and the current state, derives the state the machine should transition to next), a pointer to the Output_Column table (the table which based on the value of the inputs and the current state, derives the values for all the output variables), the Signal_Column table (the table which based on the value of the inputs and the current state, derives the values for all the signal variables). There are also two variables, Location_X and Location_Y, which are used to store the x and y coordinates of the state relative to the other states in a graphical layout of the state machine. These coordinates are used by the graphing optimizer in the process of optimizing the layout.

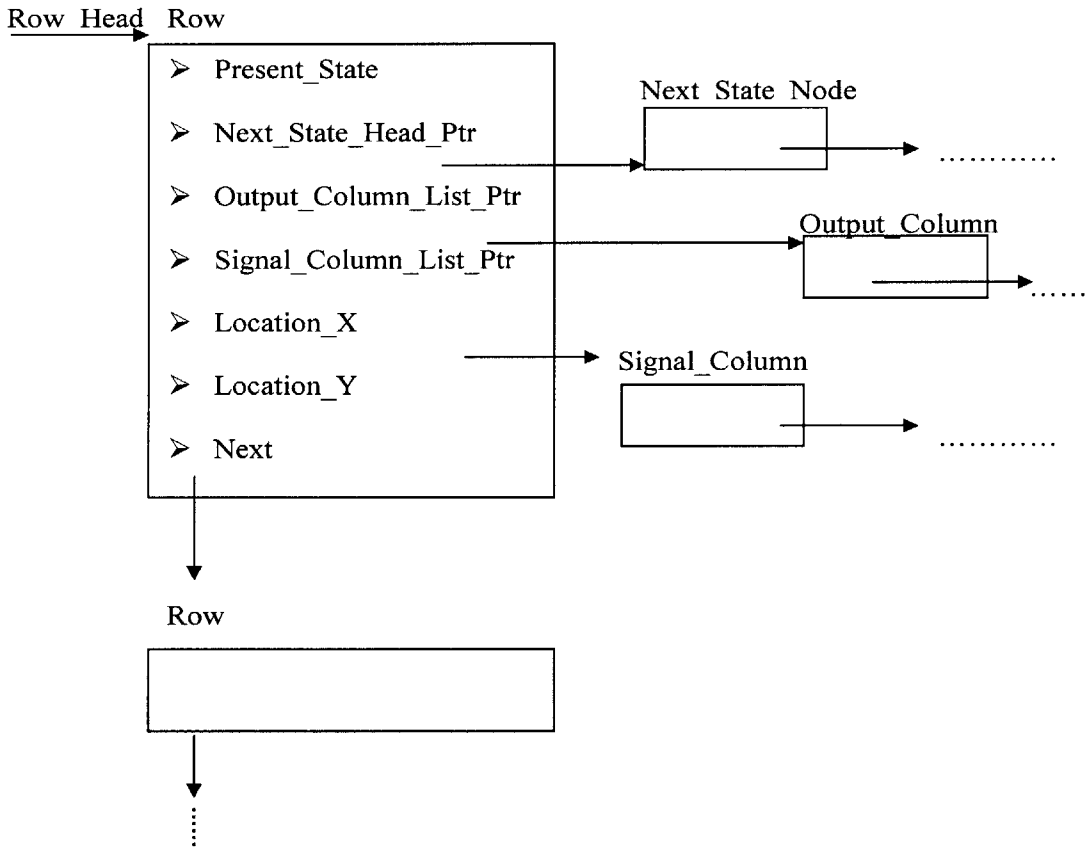


Figure 3.4. Truth Table Structure

The Next_State table which was pointed to above by the Next_State_Head_Ptr is shown in Figure 3.5. The next state table is a collection of columns where each column corresponds to a configuration of the input variables and/or signals. Each column

therefore contains a different next state depending on the input variables and signals configuration. The Input_Combinations variable points to a linked list of input/signal value nodes. The Next_State_Name corresponds to the next state that the state corresponding to this row should transition to under the Input_Combinations' conditions. The Priority variable is used to hold the priority of the emanating transition relative to the other emanating transitions. This is used when the emanating transitions are ordered in an IF/THEN construct. By using a priority scheme, one can avoid negating conditions and can instead use priorities to understand the flow of the state machine. This variable can be ignored when there is no need to order the transitions.

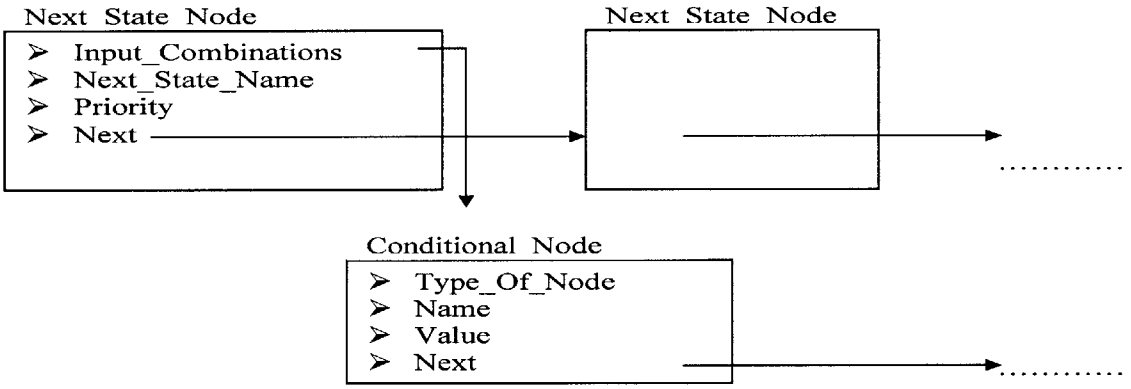


Figure 3.5. Next State Table

For example, if state A should transition to state B when (x = 1, w = 0 and f = 1) and assuming that w and x are input variables whereas f is a signal, then in the row that corresponds to state A, the Next_State_Table would look like Figure 3.6.

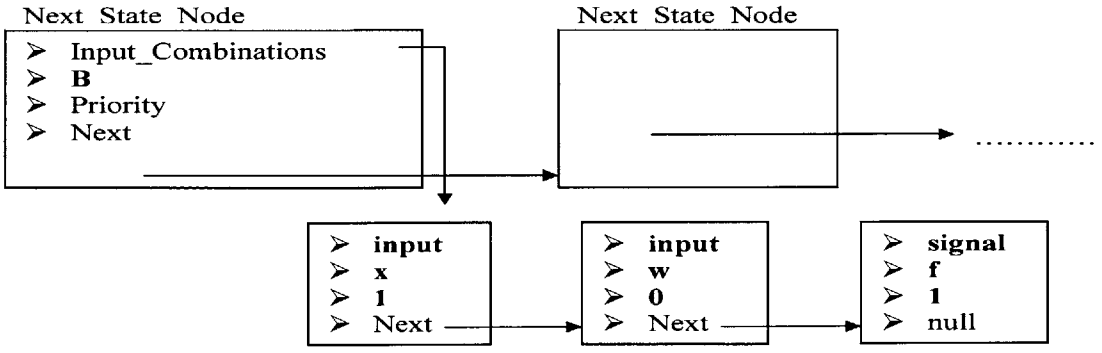


Figure 3.6. Next State Table Example

Shown below in Figure 3.7 is the output table which is pointed to by the Output_Column_List_Ptr and the signal table (Figure 3.8) which is pointed to by the Signal_Column_List_Ptr. Both tables have similar structures and so only the output table will be explained.

For a particular row, each output variable takes on a different value depending on the values of the input and/or signal variables. Unlike the Next_State table, the Output_Table has one extra hierarchical level. The reason for this extra level is that there are many output variables, each of which needs a different assignment depending on the input/signal combinations. Therefore, the extra hierarchical level is just a linked list of the different output variables present in the system. The Output_Name is used to represent the name of the output variable. The Output_Value_List_Head_Ptr points to the output value assignment table for that particular output variable.

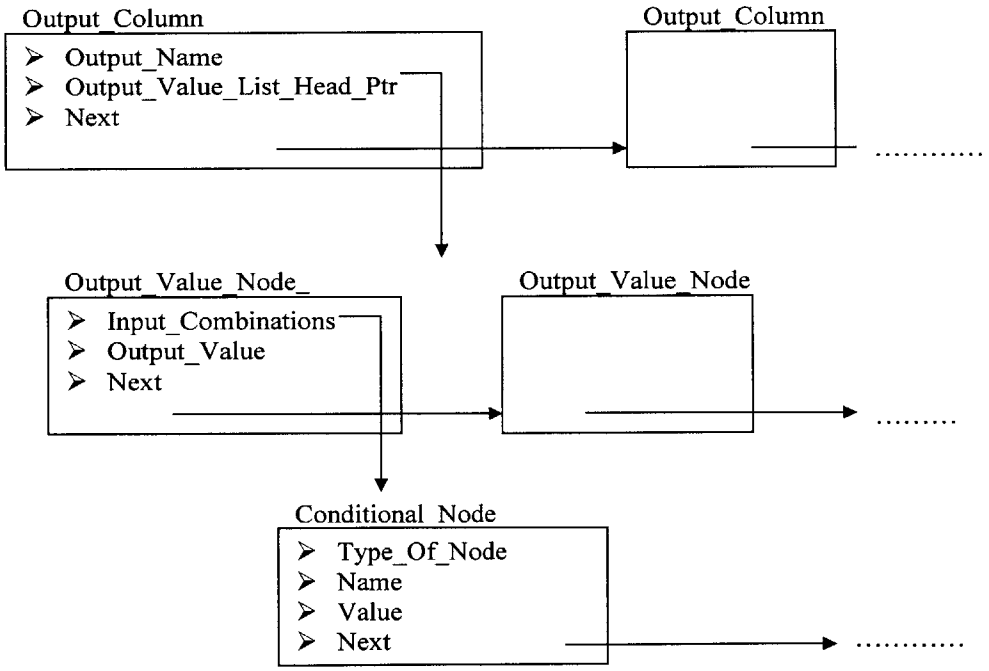


Figure 3.7. Output Table

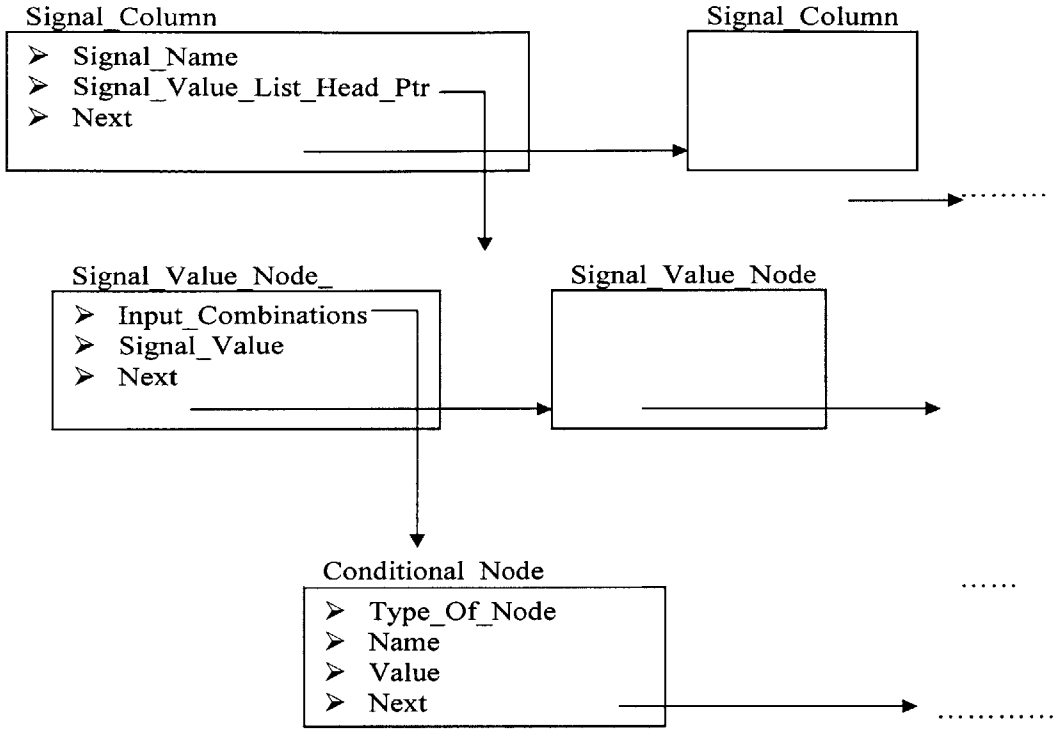


Figure 3.8. Signal Variables Table

This inner table is a collection of columns each column corresponding to a configuration of the input and/or signal variables. Each column therefore contains a different assignment value for the output variable at hand depending on the input variables and signals configuration. The Input_Combinations variable points to a linked list of input/signal value nodes. The Output_Value corresponds to the value that the output variable must take under the Input_Combinations' conditions.

For example if we had for a particular FSM, two output variables G and H, and G took the value 0 when in state A and when (x = 1, w = 0 and f = 1) and took the value 1 when (x = 0, w = 1 and f = 1), then for that particular state variable, in the row corresponding to state A, the Output_Column_List_Ptr will point to what looks like Figure 3.9.

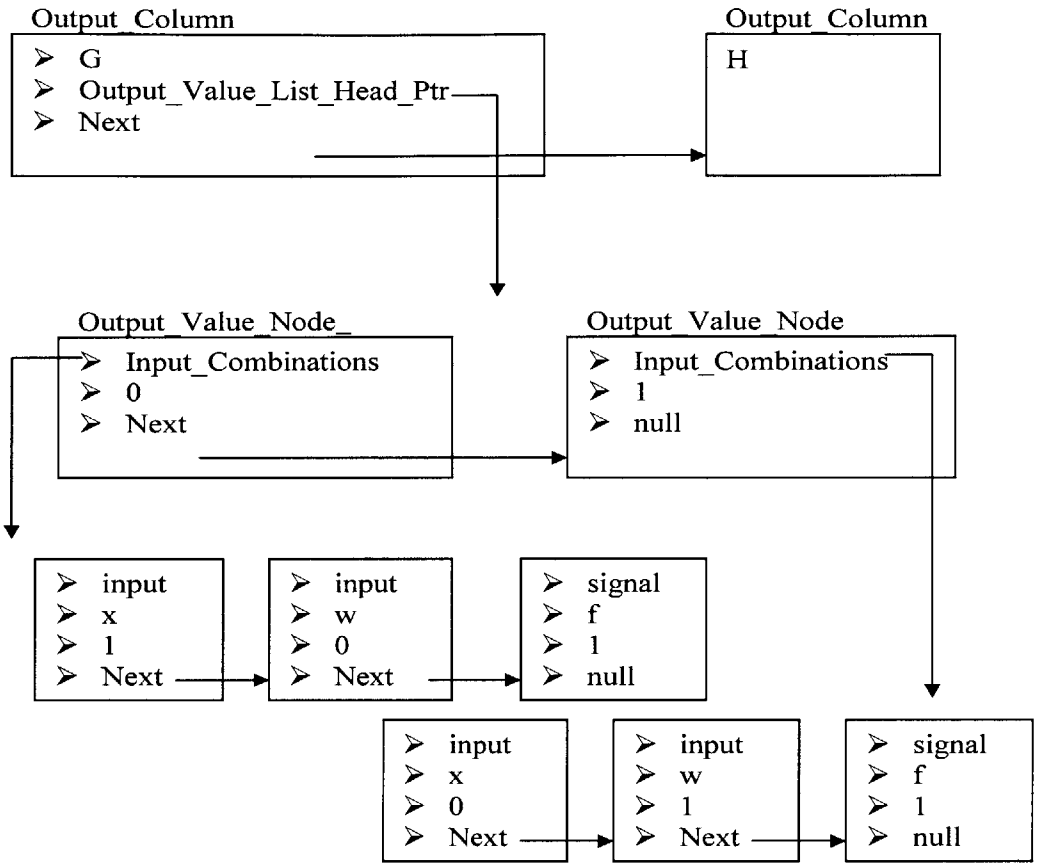


Figure 3.9. Output Table Example

- ***Parsing_Functions:*** This package contains all the functions that are used in accessing the VHDL file directly to extract the syntax. No processing other than syntactical extraction is done by functions in this package. There are several functions, used to extract tokens from the file in a different manner (i.e. some ignore spaces and semi colons, some do not ignore semi colons and in fact look for semi colons and so on).

- ***Statement_Handlers:*** This package contains functions that process different types of statements encountered in parsing the file. See section 3.2 for a detailed description of how this package achieves the model extraction.

- ***Table_Setup_Functions:*** This package is used to setup the truth tables initially when information about the states, input variables and output variables becomes available. The functions are in a package on their own to allow for modifications in the future in case the design of the underlying data structures (truth tables) is changed.

- ***Table_Write_Functions:*** This package is similar to the one above but instead writes information about transitions, output variable assignments, and signal variable assignments. This package is called by the *Statement_Handlers* package when information about assignments is found.

The two packages, *Table_Write_Functions* and *Table_Setup_Functions,* are separated from each other because their dependencies are separated as can be seen in the package dependency graph (Figure 3.2). *Table_Setup_Functions* is used to setup the data structures, while *Table_Write_Functions* is used to fill the truth tables in order to record information about the state model.

Now that the packages and their functionalities have been explained, the next section will talk about the model extraction in detail. Section 3.2 will talk about the temporary data structures used to hold the state machine information, and how those data structures are used to fill in entries in the truth table. It also discusses how the different statements are handled. A description of the modifications that are needed in order for the VAT tool to handle statements that are not currently handled is also given.

3.2 Statement Processing

There are three different VHDL statements of interest:

- definition statements
- control flow statements

- and assignment statements.

These statements are processed by using the *statement_handlers* package, and the format of the data structure used to store information gained by processing the different statements is shown in Figure 3.10.

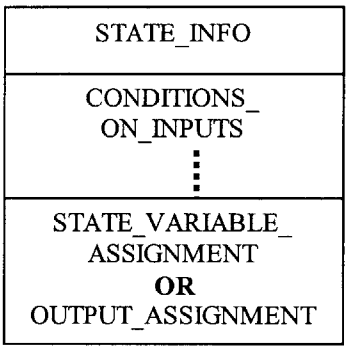


Figure 3.10. Assignment Statement Translation Structure

The STATE_INFO is the environment info which precedes the assignment statements, i.e., the assignment statement could be within a WHEN clause and is meant to occur in State X. This information is stored in the STATE_INFO section.

CONDITIONS_ON_INPUTS represents the conditions on the input/signal variables that have been placed by the preceding control statements. These conditions must exist for the assignment statements to take place. The conditions are collected from the constructs and statements preceding the assignment which specify under what conditions the assignment takes place.

The processing of the different statements will now be discussed and the way they affect the data structure is also exemplified.

Definition statements in VHDL such as the PORT statement are used to find the input and output variables that are present in the code. They specify input variables as well as output variables and internal signals (Figure 3.11). These definition statements are used to gather the input variables, output variables, and signals present in the file and are then stored in the data structures by calling the *Table_Setup_Functions* package.

```
PORT (w,f,g,h :IN
STD_LOGIC;
      x,z:   OUT   STD_LOGIC);



TYPE State-type  IS
(A,B,C,D,E);
SIGNAL  y : State-type;
```

Figure 3.11. VHDL definition statements example

Control flow statements place restrictions, such as certain values that must be held by certain input variables, which specify under what conditions the statements following the control flow statement can take place. The information gained by processing these statements is used to fill the STATE_INFO and the CONDITIONS_ON_INPUTS sections of the data structure as shown in Figure 3.12.

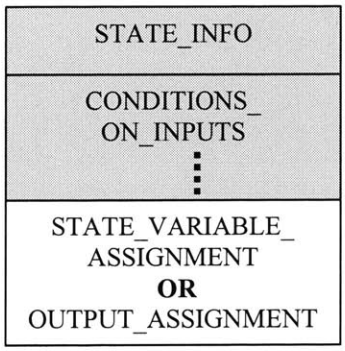


Figure 3.12. VHDL definition statements example

These conditions, end up becoming the guards on the transitions as shown in Figure 3.13.

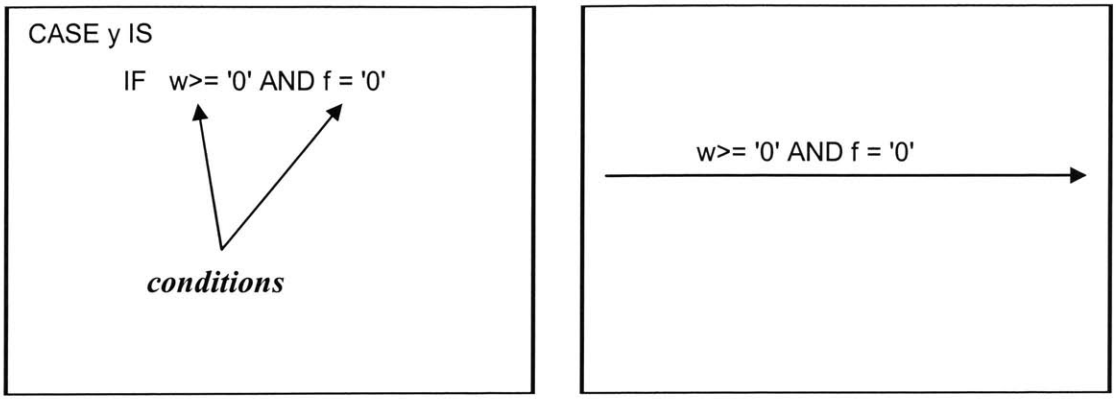


Figure 3.13. Correspondance between control flow statements and state transitions

Assignment statements can be considered to be the leafs of a tree in the recursive calls to the process_statement function and specify an assignment of a value to some variable. When assignment statements are encountered (Figure 3.14), they are used to fill in an entry in the truth table.

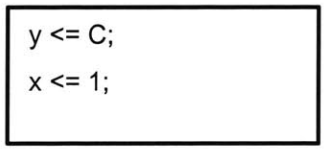


Figure 3.14. Assignment statement example

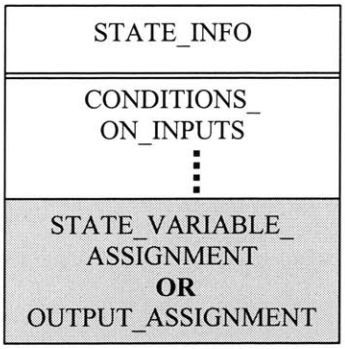


Figure 3.15. Storing of assignment statement information in data structure

There are different types of statement handlers, one for each of the above different types of statements. These handlers are in the *Statement_Handlers* package introduced in section 3.1.

A main function called "process_statement" is called everytime a new statement is encountered. The process_statement function then processes the statement and then depending on the type of statement, makes a call to a more specific handler (ex: an IF statement handler). If there is statement nesting, i.e. statements within other statements, then the specific handlers end up recursively calling the process_statement function to handle each of the nested statements. Examples of these specific handlers are the process_IF_statement handler, and the process_CASE_statement handler. The functions are used to handle IF and CASE statements respectively.

For example assume the code in Figure 3.16 existed in the input VHDL file.

```
CASE y IS
   WHEN A =>
     IF   w = '0'AND f = '0' THEN
            y <= D;
```

Figure 3.16. Example Code

If the CASE statement is not nested inside other statements, then when the CASE statement is first encountered, the process_statement function is called. Once it discovers this to be a CASE statement, it calls the process_CASE_statement function which starts processing the body of this CASE statement. Once the IF statement is encountered, the process_statement function is called again. When the process_statement function discovers this to be an IF statement, the process_IF_statement function is called. Note that the process_IF_statement function is called from the within the process_CASE_statement function (this is an example of the recursive nesting calls). The IF statement is then processed and the conditions, `w = '0'AND f = '0'`, are recorded in the in the CONDITIONS_ON_INPUTS section of the temporary data structure. The IF statement then continues to be processed and as soon as the "`y <= D;`" statement is encountered, the process_statement function is called again. This time the process_statement function discovers the statement to be a state assignement statement

and so makes a call to the process_STATE_ASSIGNMENT_statement function. That function then stores the statement, "y <= D", in the STATE_VARIABLE_ASSIGNMENT section of the temporary data structure in Figure 3.10. The temporary data structure (Figure 3.17) is then used to record an entry in the truth table which corresponds to a state transition which specifies that when state machine y is in state A and the inputs w= '0' and f = '0', then the state machine transitions to state D.

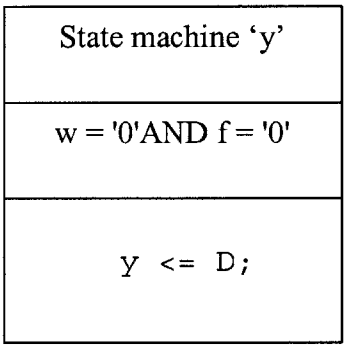


Figure 3.17. Example of filled data stucture

If there were more statements nested inside the IF statement, then they would also be handled recursively by calls to the process_statement function. Since there are no such statements, the process_IF_statement function ends and returns control to the process_statement function. The process_statement function than recurses one step back to the process_CASE_statement function which then continues processing the other WHEN clauses within the CASE statement.

Therefore, recursively, each statement in the body of the code is processed until the inside most control function is reached. This allows us to remove any assumptions on how an FSM is represented in VHDL (ie. there is no assumption that an FSM always starts with a CASE statement and then has a set of IF statements inside, or any other structure that is commonly used).

When a data structure such as the one shown in Figure 3.17 is filled (ie. after we reach an ASSIGNMENT_STATEMENT), it is used to fill in an entry in a truth table (Figure

3.18). The location of the entry depends on the STATE_INFO and the CONDITIONS_ON_INPUTS variables.

Current State/ Inputs:	Next State w=1 y=0	w=0 y=1	w=0 y=0	w=1 y=1	Output1	Output2
A	B	A	A	B		
B	A	A	A	B		

Figure 3.18. Truth Table Example

Once the truth table is filled, an intermediate format for the state model is reached. A specialized translator is then used to transform this intermediate representation into a notation specific to the target model checker.

As a proof of concept, a translator which generates an XML file containing the formal model information was designed and implemented. XML is the notation currently accepted by the model checker Uppaal. Another model checker, might require a notation different from XML. Spin for example accepts Promela as an input notation, and hence a specific translator from truth tables to Promela would be constructed. This separates the extraction of the model from the representation specific to the target model checker. This allows us to achieve the one-to-many (one VHDL specification to many model checking notations) relationship we set out to reach.

The problem with the above assignment statement structure, Figure 3.10, is that it assumes that state variable transitions are independent of other state variables in other FSMs. However, if synchronization is presented in the code, in the sense of different automata communicating and synchronizing, that ceases to be the case.

Synchronization, a coordination with respect to time, refers to a guard on a transition that is dependent on some other transition in another FSM. VHDL does not contain an explicit representation for synchronization. In VHDL, synchronization is present in an implicit way, whereby transitions are dependent on some signal in shared memory. When

two transitions in two different FSMs depend on that same variable, synchronization is achieved. The dependence of one signal variable, CURRENT_STATE, on another, STATE_VARIABLE_2, is shown in Figure 3.19.

```
BEGIN
  PROCESS(Resetn, Clock)
    BEGIN
      CASE CURRENT_STATE
        WHEN A =>
      IF (STATE_VARIABLE_2 = J)
        NEXT_STATE = B;
      END IF;
       END CASE;
  END PROCESS:
END Behavior;
```

Figure 3.19. Implicit Synchronization in VHDL

A modified translation structure was designed to account for the increased complexity as illustrated in Figure 3.20. In this new structure, an extra variable, CONDITIONS_ON_OTHER_STATE_VARIABLES, is added to account for the dependence of the next state transition on the conditions of other state variables.

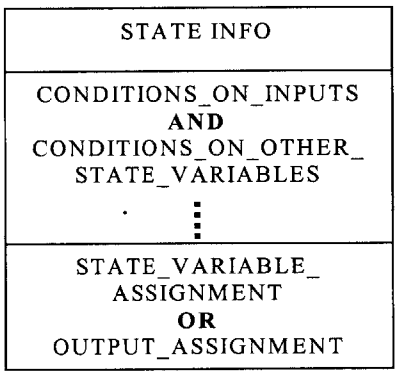


Figure 3.20. Modified Assignment Statement Translation Structure

To account for the implicit synchronization mentioned before, the original truth table format had to be modified. Signal values were added in the next state columns and output columns where we initially only had CONDITIONS_ON_INPUTS. The modifications to the next state columns is shown in Figure 3.21.

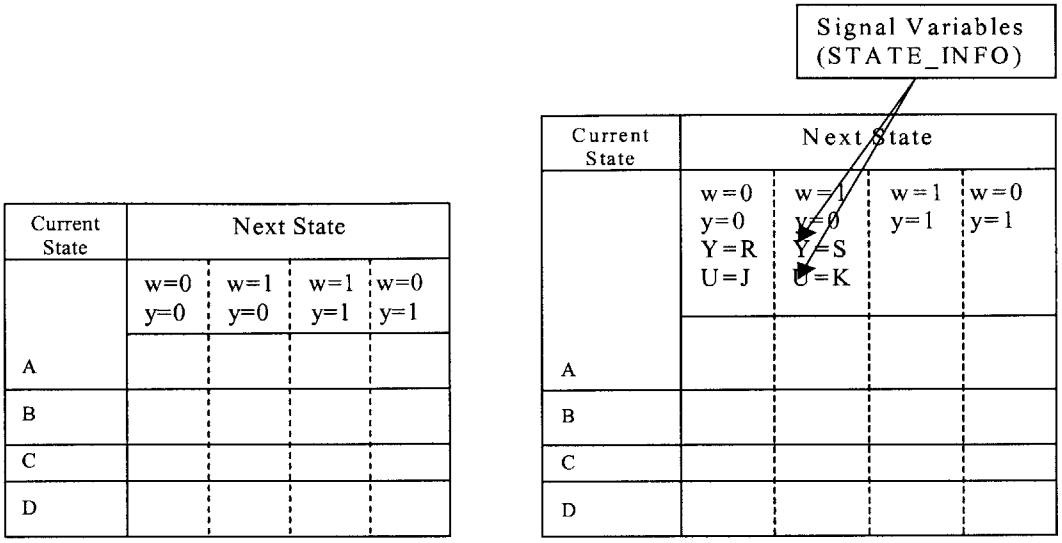


Figure 3.21. Truth Table Modification ***(Inputs are shown in small letters whereas States are shown in capital letters)***

The interpreter ignores all VHDL syntax which is irrelevant to describing finite state machines. Below is an extract from the table of contents of the IEEE 2002 VHDL reference manual [1076]. It lists the sequential statements, Table 3.1, that are provided by the VHDL language. In the right column, the statements that are handled by the tool are flagged as 'HANDLED' and the ones that are not, are flagged by 'NOT HANDLED'.

The sequential statements that are not handled have not been used to describe FSMs in the multitude of examples that have been reviewed. Concurrent statements are not handled, because they are rarely ever used to describe state machines in VHDL. As well, many of the constructs ignored deal with issues such as pin assignments and other non-contributive constructs.

The statements that are not handled can be handled by modifying the *Statement_Handlers* package. For every new construct that needs to be handled, a corresponding function needs to be coded that handles the construct. That function must also be able to handle the information provided by the construct and attach it to the conditional list that accumulates all the different conditions. Also the process statement

function in the *Statement_Handlers* package needs to recognize the syntax used by the construct and to make a call to the new function when the construct is encountered.

Sequential statements	VAT status
Wait statement	NOT Handled
Assertion statement	Handled
Report statement	NOT Handled
Signal assignment statement	Handled
Variable assignment statement	Handled
Procedure call statement	NOT Handled
If statement	Handled
Case statement	Handled
Loop statement	NOT Handled
Next statement	Handled
Exit statement	Handled
Return statement	Handled
Null statement	Handled

Table 3.1. Sequential VHDL statements and VAT status

4. Visualization

An added benefit from doing this VHDL-FSM transformation might possibly be achieving enhanced readability. However, it was apparent early on in the development of the tool, that as the size of the code would grow and the corresponding number of states would increase, the enhanced readability projected might not actually be realized.

The problem that arises is that of crossing edges. Since a finite state machine is depicted as nodes and edges, a random placement will result in some crossing edges. As the number of states and hence edges grow, a large number of these crossings can ensue. These large number of crossing edges cause the graphical representation of the state machine to become illegible (Figure 4.1). Before a solution to this problem is presented, a fair discussion of what readability is and how legibility derives from it is warranted.

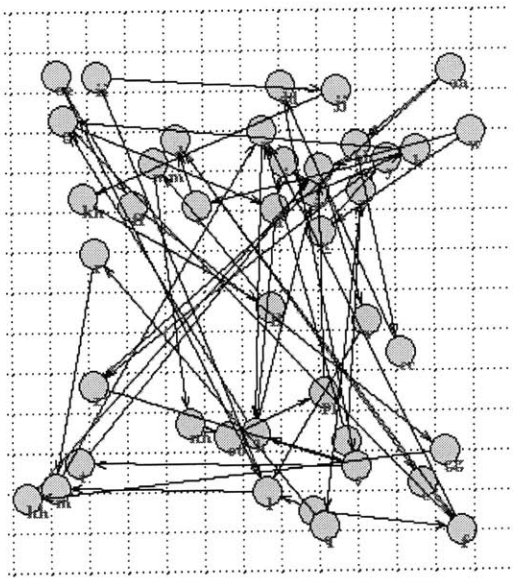


Figure 4.1. Cob-Web problem

Readability of an object can refer to a set of criteria which together improve the object's ability to perform its intended duty. Readability of a specification language could refer to the union of many qualities [Har00]. Examples of these qualities are:

- conciseness
- legibility
- comprehensibility
- interestingness

Conciseness refers to the wording of the object. Overuse of words, and complicated descriptions adversely affect readability, and hence conciseness is one of the qualities sought after when attributing a readability grade level. Legibility refers to the quality of an object that allows a human being to clearly make out the terminology. This is mostly an issue that arises due to graphical inefficiencies. Comprehensibility refers to the ability for one to understand what the object is trying to represent. This is a matter of lucidness and well representation on the object's part. Interestingness refers to the ability of the information the object is representing to be interesting. This is obviously dependent on the particular audience that will be doing the reading.

Software readability refers to the legibility, or the easiness by which the software is understood, both by reading the documentation and by reading the code [Han98]. It is one thing to have a correct representation, but if that representation is unusable because it is unreadable, then the representation if of no use.

It is common to confuse readability and usability [Han98]., however, while usability pertains to the end user and the ability of the product to perform its intended function, readability applies to the maintenance programmer and his/her ability to apply modifications. A deficiency in usability creates an impact in the short term when the consumer dislikes the product. Readability however, affects future generation of the code and the maintenance stage, and is not a consumer good for sale.

In a software product's life cycle, program modification is costly and consumes a large share of life cycle [EM82]. Since programs can be characterized as solutions to some demand, the programs will need to be changed to accommodate changes in what the program is needed to solve. The problem that frequently arises during program modification is that they are found to be extremely difficult to read and understand. This causes the price of modification to jump tremendously.

Another aspect to readability as pertaining to software, is the readability of the formal methods used to specify the code. A study by Marc Zimmerman [ZLL02] shows that different formal methods can be attributed different levels of readability due to the way these formal methods appeal to readers. The conclusion was reached by running a set of experiments where the subjects had to respond to questionnaires. The responses were used to evaluate the difference in readability between the different formal specification languages.

The tool being discussed addresses the issues of readability by transforming VHDL specifications to FSMs. Unlike the aspects of readability introduced above, a new dimension arises when we try to compare the readability of the formal model abstraction to that of the actual code. In this case the readability issue that arises is not the level of readability that is attributed to the code, but the difference in readability between the original VHDL code and the FSM formal model that is generated by the tool.

For such a comparison to be valid, the FSM and the original code have to be two different representations of the same information. Indeed they are not, the VHDL code contains more information than the FSM, especially since in generating the FSM we are abstracting the program away and just extracting the information that pertains to states and transitions. However, if it the state machine behavior is the most important aspect of the system described by the code, then we can compare the VHDL code and the FSM as if they were two different representations of the same information; i.e. the state machine behavior.

Now that we have developed a basis for comparison, it is possible to talk about the difference in readability between the VHDL code and the FSM. Since we have reduced

the VHDL code and the FSM representation to two different representations of a certain aspect of the system, we can compare the levels of readability each offers. This comparison will allow us to see if the FSM representation can offer us a new method for viewing the state machine information in a more readable format.

This increased readability from the FSM representation is a projected readability, and has not been tested. It was actually realized as a potential benefit after the tool was developed and was not one of the initial goals set out. An experiment would have to be conducted along with some supporting formal arguments to show that, indeed, increased readability can be achieved. That is left for possible future work.

The graphing optimizer is designed to insure that if indeed, increased readability is achieved by a transformation to an FSM representation, then the crossing edges problem would not hinder that.

The graphing optimizer reduces the number of edge crossings in a diagram of states (nodes) and transitions (edges). With a clearer topology, reading such a diagram becomes easier and hence adds to the readability of the FSM. Since the qualities mentioned above are important attributes of a good specification, they must also be attained by our FSM depiction of that specification (VHDL).

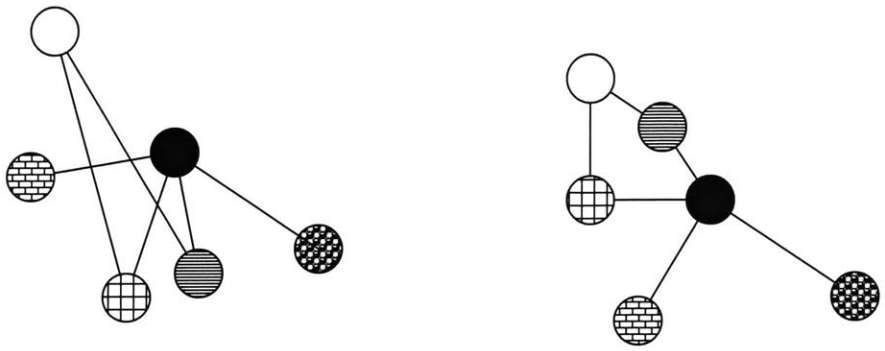
The issue that arises with the crossing edges problem is that it reduces legibility, which in turn would reduce readability. Therefore, whether or not we can prove that an FSM also gives us the added benefit of increased readability, we needed to address any factors that can adversely affect it.

In doing so, a modified version of the spring embedder algorithm was designed and implemented and was used to improve the visualization by reducing the number of crossing edges.

4.1 Spring Embedder Algorithm

Planar graphs are graphs that can be drawn without edge crossings. It is possible to test for planarity, and there are several algorithms to draw a planar graph without crossing edges.

The two graphs in Figure 4.2 are different layouts of the same graph. The graph on the left contains three edge crossings, and the one on the right has zero crossings.



(a) (b)

Figure 4.2. The graph in (a) is planar. It is redrawn in (b) without crossing edges

Because crossings significantly decrease the readability of a drawing, if a graph is not planar, it is best to draw it with as few crossings as possible. Researchers have equipped us with several layout algorithms to solve this problem. One interesting class of layout algorithms are called "force-directed placement" algorithms. These algorithms simulate nodes that repel themselves while being attracted by edges between the nodes which function as springs. This results in nodes with edges between them appearing close to one another while disconnected nodes appear distant.

The graphing optimizer is based on the "Spring Embedder Algorithm" [FR91], a force-directed placement algorithm. The Spring Embedder algorithm uses attractive and repulsive forces to move the nodes into a stable topology. The idea of the algorithm is the one of simulating a system of mass particles. The vertices simulate masses repelling each

other and the edges simulate springs with attracting forces. The algorithm tries to minimize the energy of this physical system and in doing so finds the configuration with the minimum number of crossing edges since that configuration corresponds to the one of least energy.

The algorithm works by attaching a spring (represented by an edge) between every two vertices. Each spring is then assigned a natural length. Springs between vertices which are adjacent (already have an edge between them) are assigned a length of some small value Φ. Springs between vertices which are disjoint are assigned a much larger natural length Γ. Choosing Γ affects the strength with which non-adjacent vertices push away from each other. The larger Γ is, the longer is the spring, and hence the more it will try to push the vertices away from each other. However, if Γ is chosen to be too large, then Φ will be relatively small and hence the attractive force between adjacent vertices will not be enough to pull the vertices together.

Once all the edges are assigned springs between them, and the natural lengths of the springs are determined, the algorithm proceeds to minimize the total energy of the overall configuration. The energy is minimized when springs are least extended or compressed. Hence adjacent nodes which are far away try to come close together because the spring between them is extended (length > Φ). On the other hand, non-adjacent nodes which are close together try to push apart due to the long spring between them which is contracted (length < Γ) . At every iteration of the algorithm (Figure 4.3), the attractive and repulsive forces between the different vertices are calculated and the summation of all the forces on any single vertex is used to move that vertex in the direction of the resultant force. The iterations continue until a configuration of minimum energy is reached. The number of crossing edges is reduced since crossing edges increase the energy of the system. The energy of the system can be calculated by using the following summation:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \tfrac{1}{2} k_{ij} \left(\left| p_i - p_j \right| - l_{ij} \right)^2 \quad (1)$$

Particles on a plane p_1, p_2, ... p_n represent vertices v_1, v_2, ... v_n. l_{ij} is the natural length of the spring between v_i and v_j and is defined as $l_{ij} = L \times d_{ij}$, where L is the desired length of the edge in the layout and d_{ij} is the shortest length path between v_i and v_j in the graph. The strength of the spring between v_i and v_j is k_{ij}, and is defined as $k_{ij} = K / d_{ij}^2$, where K is a constant [Sim96].

The algorithm is summarized by the following pseudo-code.

1. compute d_{ij} for $1 \leq \neq \leq i\ j\ n$;
2. compute l_{ij} for $1 \leq \neq \leq i\ j\ n$;
3. compute k_{ij} for $1 \leq \neq \leq i\ j\ n$;
4. initialize p_1, p_2, p_n
5. while (max $\Delta_i > \varepsilon$)
6. let p_m be the particle satisfying $\Delta_m = \max \Delta_i$;
7. while ($\Delta_m > \varepsilon$)
8. compute δx and δy ;
9. $x_m \leftarrow x_m + \delta x$
10. $y_m \leftarrow y_m + \delta y$
11. end while
12. end while

Figure 4.3. Spring Embedder pseudo-code [Sim96]

In the above pseudo code, δx and δx, are the incremental changes in the positions of the vertices. They are calculated by using the resultant force An added benefit from the imaginary springs of length Γ between the non-adjacent vertices is that they limit the overall size of the layout.

4.2 Graphing Optimizer

A modified version of the spring embedder algorithm was derived and implemented as a graphing optimizer for our tool. The reason for using a modified version is the extensive computational demands of the original spring embedder algorithm. Derivatives

are used to make incremental changes in the positions of the masses (nodes). These derivatives include divisions and are performed $O(n^2)$ number of times where n is the number of nodes. In order to reduce the computational complexity, the modified version reduces the number of derivatives required by taking derivatives with the surrounding nodes only.

The difference in this modified version is that it removes some calculations that do not contribute significantly to the displacements. For example, in step 8 (Figure 4.3), the calculations with the distant non-adjacent nodes are not performed due to their small attractive/repulsive effect; the springs are long for non-adjacent nodes and since they are distant, the force exerted by the springs is minimal. Computing the δx and δy is costly due to the divisions that need to be performed. By reducing some of these calculations, the computation time is reduced.

The code used to generate Figure 4.1 is shown again below in Figure 4.4 after running the code through the graphing optimizer.

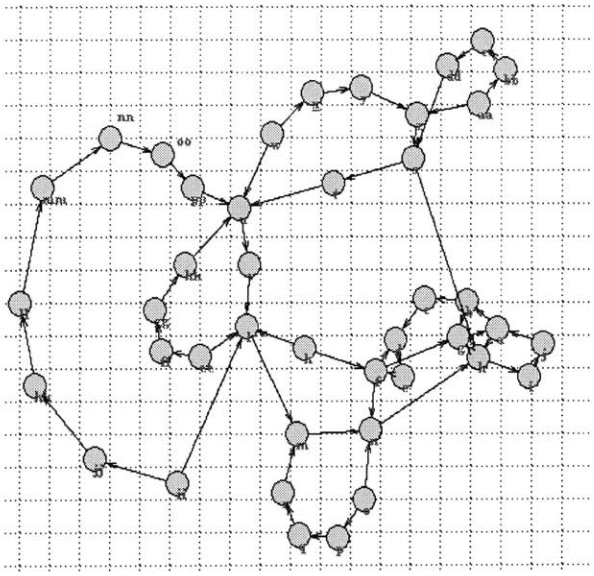


Figure 4.4. Cob-Web problem from Figure 4.1 solved

An additional benefit that was only realized after implementation is the symmetry that results from using the spring embedder algorithm. As will be shown in chapter 5, a

case study revolved around showing these symmetries that arise and the advantage that they provide to the programmer in understanding the system behavior.

The bottleneck for the tool in terms of time consumption lies in the graphing optimizer. As the FSMs grow in size, i.e. number of states, the graphing optimizer takes longer to calculate the optimal placements. This however is not an obstacle when viewed from the vantage point of the Gurkh-project as a whole. The graphing of the states and their transitions, and more specifically optimizing their placements is an aesthetic luxury used to increase the comprehensibility of the diagram. However, when incorporated into the Gurkh-project, this luxury is no more necessary and hence the bottleneck is removed. Also, the fact that model-checking is so computational expensive, the $O(n^2)$ cost accounted for by the graphing optimizer becomes insignificant when seen from the model checking process as a whole.

5. Tests and Results

This section presents an example of a transformation followed by two case studies performed and the results that were attained.

5.1 Example 1

The following VHDL code (Figure 5.1) is a section extracted from a VHDL file that was used for a test.

The code is part of a VHDL file representing an FSM which describes a memory read. After running the full VHDL code through the tool and then viewing the resulting XML file using Uppaal, the representation shown in Figure 5.2 was the result.

In the example VHDL file, lines 21 to 24, we can see that when in state Read, if the input Bus2IP_CS is 0 and ReadB is also 0, then we leave state Read and go to state Read2. As well as transitioning, the signal Bus2IP_CS and LoadA both change values to 0 and 1 respectively. This is seen very clearly in the FSM shown in the Uppaal snapshot. We now can using Uppaal, assert certain timing constraints and make sure that our VHDL code adheres to them. For example, we can test whether or not the system will experience deadlock during runtime or whether certain states are reachable.

```
1 BEGIN
   PROCESS(Resetn, Clock)
   BEGIN
      IF Resetn = '0' then
5        y <= A;
      ELSIF (Clock'EVENT AND Clock = '1') then
        CASE state IS
           WHEN idle =>
               state <= read;
10         WHEN read=>
               IF (RxAv='0') then
                   ReadB<='0';
               end if;
      IF (IP2Bus_ack='1' AND Bus2IP_CS='1') then
15      Bus2IP_CS<='0';
        txdata_reg(3)<=IP2Bus_Data(24 to 31);
        txdata_reg(2)<=IP2Bus_Data(16 to 23);
        txdata_reg(1)<=IP2Bus_Data(8 to 15);
        txdata_reg(0)<=IP2Bus_Data(0 to 7);
20    end if;
      IF (Bus2IP_CS='0' AND ReadB='0') then
        Bus2IP_CS<='0';
        LoadA <= '1';
        state <= read2;
25    ELSE
        state <= read;
      end if;
           WHEN read2=>
      IF (TxBusy='1') then
 30     LoadA <= '0';
        state<=read3;
      ELSE
        state<=read2;
                  end if;
```

Figure 5.1. Section of VHDL test file

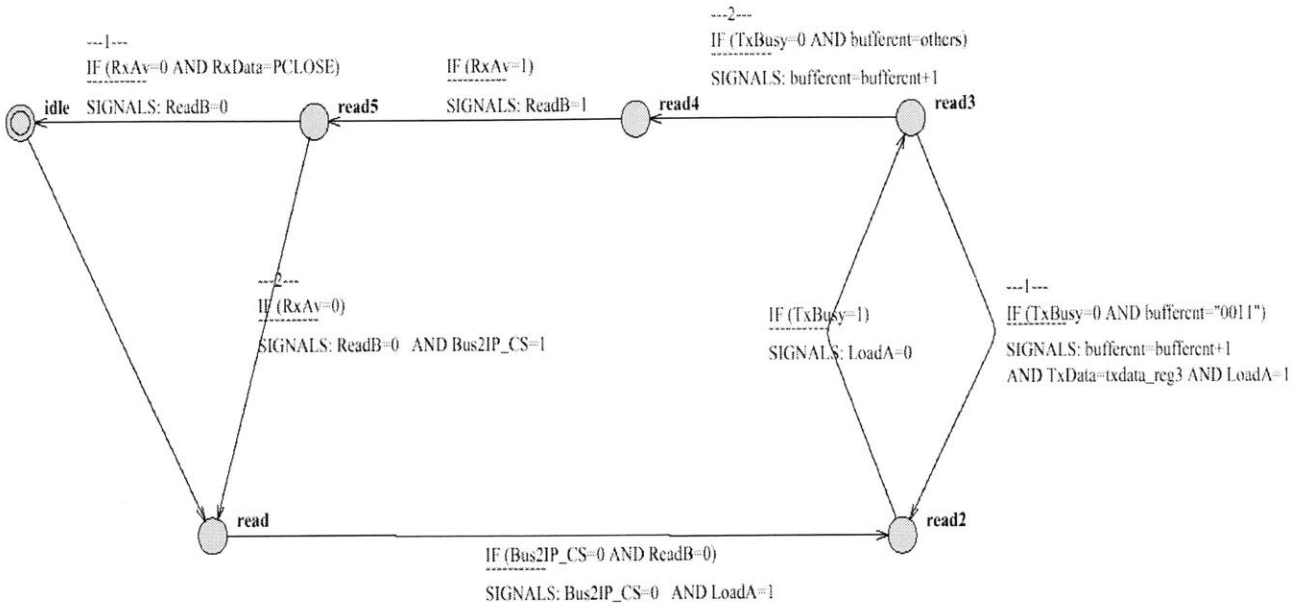


Figure 5.2. Resulting FSM representation of VHDL test file in Figure 5.1

5.2 Test Case 1

In this test case, we are trying to show how model checking can be applied on the extracted models. A VHDL program is used that contains three state machines shown in Figures 5.3, 5.4 and 5.5.

One of the state machines, Figure 5.3, is a read_state_machine and is used to read something information from memory. The state machine first checks to see if the bus is free and if so, takes control and issues a read command.

```
CASE y IS
      WHEN RESET =>
            IF   read   = '1'AND erase = '0' THEN
                 y <= PREPARE_FOR_READ;
           ELSE y <= RESET;
           END IF;

      WHEN PREPARE_FOR_READ =>
          IF bus_clear = '0' THEN
               y <= READ; bus_Clear <= '1';
          ELSE y <= PREPARE_FOR_READ;
          END IF;

      WHEN READ =>
         y <= CLEAR_BUS;
          read <= '0';

      WHEN CLEAR_BUS =>
          y <= RESET;
          bus_Clear <= 0;
END CASE;
```

Figure 5.3. Read FSM

A second state machine, Figure 5.4, is an erase_state_machine and does a similar as the read_state machine but instead attempts to erase information from memory.

```
CASE q IS
        WHEN RESET =>
           IF   erase   = '1'AND read = '0' THEN
                  q <= PREPARE_FOR_ERASE;
           ELSE q <= RESET;
           END IF;

        WHEN PREPARE_FOR_ERASE =>
           IF bus_clear = '0' THEN
                  q <= ERASE; bus_clear <= '1';
           ELSE q <= PREPARE_FOR_ERASE;
           END IF;

        WHEN ERASE =>
          q <= CLEAR_BUS;
          erase <= '0';

        WHEN CLEAR_BUS =>
          q <= RESET;
          bus_Clear <= 0;
END CASE;
```

Figure 5.4. Erase FSM

The third state machine, Figure 5.5, is the bus controller and decides which of the other two state machines gets control of the bus.

```
CASE user IS
        WHEN RESET =>
           user <= Erasing;
             erase <= '1';

        WHEN Erasing =>
            IF erase = '0' THEN
                user <= Reading; read <= '1';
            ELSE user <= Erasing;
            END if;

         WHEN Reading =>
            IF read = '0' THEN
                  user <= Erasing;
                  erase <= '1';
            ELSE user <= Reading;
            End if;
END CASE;
```

Figure 5.5. User FSM

The VHDL code for these three state machine was run through the VAT tool and the formal model was presented to Uppaal. The generated FSMs are shown in Figures 5.6-5.8 below.

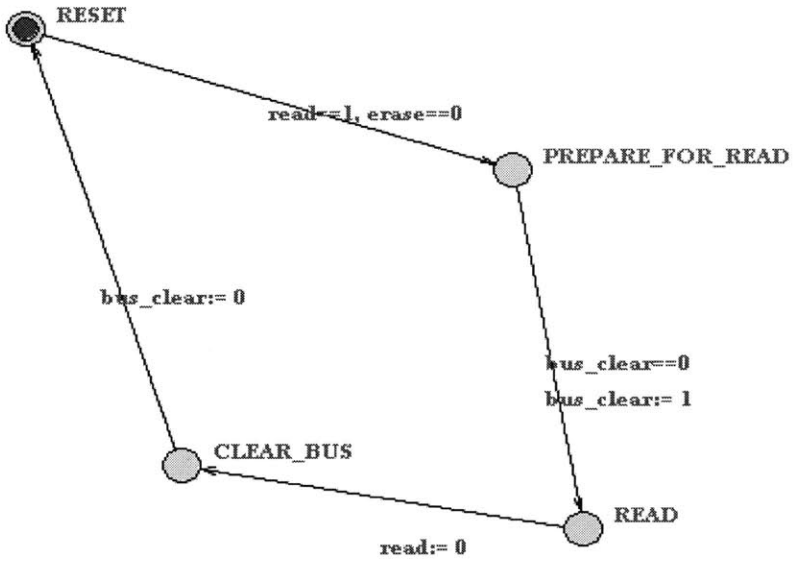


Figure 5.6. Read FSM

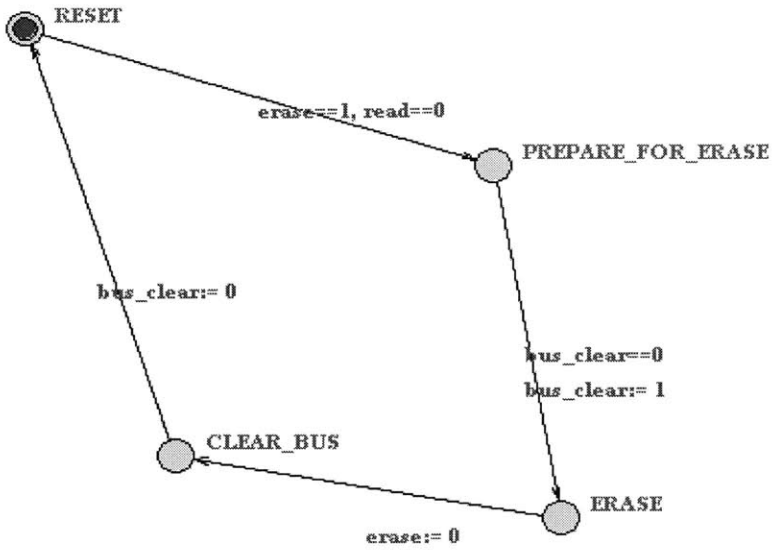


Figure 5.7. Erase FSM

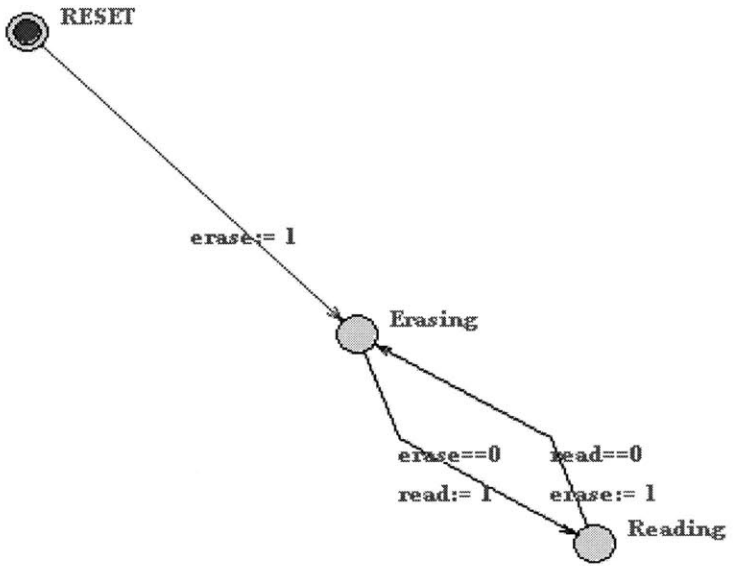


Figure 5.8. User FSM

The Uppaal verifier was then used to check whether or not certain assertions hold. A snapshot of the verifier screen is shown in Figure 5.9. The four assertions will now be explained in detail to give the readers an idea of the type of assertions that can be verified.

All the statements are modified versions of propositional logic. The A[ ] and E<> symbols are quantifiers and represent the "for all" and the "there exists" quantifiers.

- A[ ] not deadlock: this assertion states that for all possible states the different state machines can be in, there will not be deadlock. This is an extremely important property to verify especially in embedded systems where deadlock could mean the halting of a mission and the loss of reliability in the system.

- E<> y1.READ: This assertion uses the "there exists" quantifier to check where or not state machine y is ever in the READ state. We might want to use such assertions to see whether certain states that should be reached are indeed reachable, or to find states that are unneeded states which can be discarded (a form of optimization).

- E<> (y1.READ and q1.ERASE): this will check whether it will ever be the case (notice unlike for the A[] quantifier, the term never is not used) that state machine y will be in the READ state and machine q will be in the ERASE state. As one can see, this is an important property to test for in this particular system. The verifier will then explore the state space and then either verify that the assertion holds or give a counter example to why it does not.

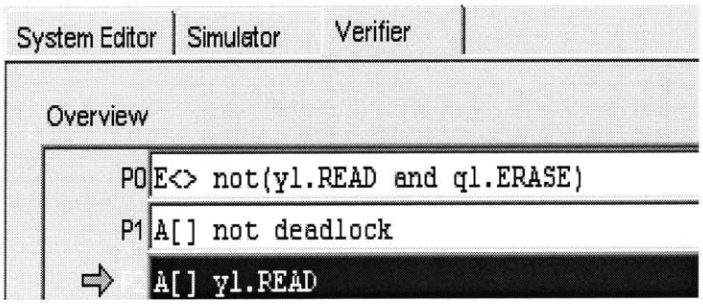


Figure 5.9. Snapshot from Uppaal Model Checker

5.3 Test Case 2

In this test case, the proper functionality of the tool was tested. The test subject was a person who had been designing and implementing VHDL specifications for some time (almost 6 years). The test subject had developed some code for a PowerPC port. The VHDL code was run through the tool, and then the formal model was viewed using Uppaal. The test subject was able to verify that the model represented the code exactly. He also confirmed that it was easier to look at the different sections of the finite state machine then to look at the code to find errors.

In a set of test runs, the code was modified to seed errors without notification of the test subject. The test validated that the seeded errors could be caught by inspecting the formal model.

There was also another benefit that was realized due to the algorithm we have chosen for the graphing optimizer (Spring Embedder). As mentioned in section 4, the algorithm works by bringing nodes that have edges between them closer, and pushing nodes that do not have edges between them away from each other. This causes clusters of states as shown in Figure 5.10 below.

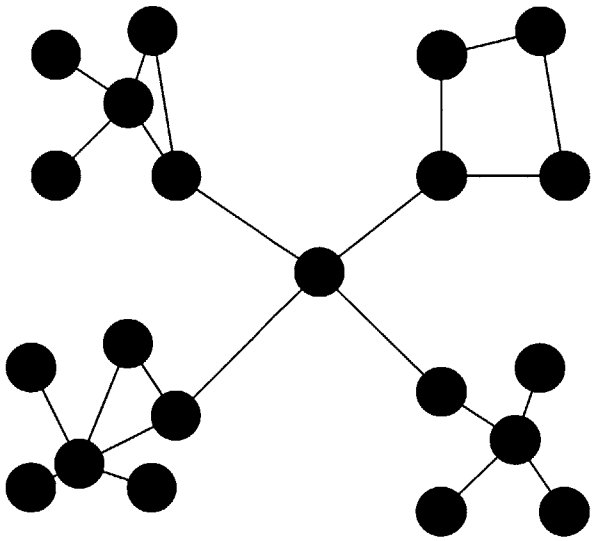


Figure 5.10. Cluster of states

This symmetrical distribution causes states that have transitions between them to group together, separated from other clusters. Therefore, even as the systems we are formalizing become more complex and hence have more states, this symmetry causes the formal model to be organized in such a fashion that we have hierarchal distributions which are easy to read. This was realized in this experiment, when the code that was used was large, but the formal model was still very readable. The test subject had to scroll to different sections of the code, where states that were closely related were clustered together. When that part of the formal model was in focus on the computer screen, it could easily be analyzed separate from the other clusters.

6. Conclusions

Embedded systems have become an integral part of the systems we use today. These types of systems are constrained by both stringent time requirements and limited resource availability. Traditionally, high-integrity embedded systems operated on well understood hardware platforms. The emergence of inexpensive FPGAs (Field Programmable Gate Arrays) and ASICs (Application Specific Integrated Circuits) as operational platforms for embedded software, has resulted in the system developer having to verify both the hardware and the software components. The stringent processes used over the system development lifecycle have to be augmented to account for this paradigm shift. One possible approach is to create a homogenous formal model that accounts for both the hardware and the software components of the system.

One of the goals of the thesis was to provide the system developer with verification and validation approaches other than testing. VAT provides two alternative techniques for system verification and validation: model-checking and manual inspection. The finite state machine representations that VAT extracts from VHDL specifications can be model-checked using Uppaal, and the graphing optimizer provides a visualization of the formal model for manual inspection.

One of the limitations of existing tools is the lack of modularity and their inability to adapt to changing user requirements. The design of the VAT tool has accounted for this need for extensibilitiy. VAT can be easily extended to other hardware description languages, as syntax specific information is present only in the primary stage: the tokenizer. The rest of the steps in the transformation process are specific to extracting a formal model, storing it in an intermediate format, and then grooming it to be fit for the target model checker. VATs transformation process is transparent and easy to

understand. This enables both the academic and industrial communities to further extend the tool and its capabilities to meet both research as well as development goals.

The tool was tested with various VHDL files as inputs. The test files included actual software taken from different applications which included port interfaces, bus controllers and other similar applications. These tests are important because they increase the confidence attributed to the proper operation of the tool.

In the overall context of the Gurkh project, the VAT tool provides a critical link that performs the formal model extraction. The formal model of a VHDL specification in combination with the monitoring chip, can greatly enhance the safety of the whole system by assuring that errors that would otherwise go on undetected will be caught in due time.

Future work will include using program analysis to automatically derive assertions from the VHDL specifications. This automated analysis will act as a supplement to the system developer specified assertions. Research has to be carried out to determine the right combination of automatically generated assertions and system designer generated assertions.

7. References

[1076] IEEE Std 1076 VHDL Language Reference Manual. 2002. IEEE Computer Society.

[1364] IEEE Std 1364 Verilog hardware description language (HDL) standard. 2001. IEEE Computer Society.

[ABC82] W. Adrion, M. Branstad, J. Cherniavsky. 1982. Validation, Verification, and Testing of Computer Software. ACM Computing Surveys (CSUR), Volume 14 , Issue 2 (June 1982), pages: 159 - 192

[AC95] Laurent Arditi, Helene Collavizza. Towards Verifying VHDL descriptions of Processors . Proceedings of the conference on European design and Automation, pages: 414- 419

[ACD90] R. Alur , C. Courcoubetis, D. Dill. 1990. Model-checking for real-time systems. In Proceedings of 5th LICS. IEEE Computer Society, New York, pages: 414–425.

[Ada95] Ada 95 Reference Manual (Language and Standard Libraries), revised international standard (ISO/IEC 8652:1995)

[AIA04] AIAAJournal. 2004. Journal of Aerospace Computing, Information, and Communication.

[AJL+99] L. Asplund, B. Johnson, K. Lundqvist, and A. Burns. 1999. Session summary: The Ravenscar profile and implementation issues. The 9th international Real-Time Ada Workshop (IRTAW), Ada Letters XIX(2). Pages: 12-14.

[AL03] L. Asplund, K. Lundqvist. 2003, The Gurkh Project: A Framework for Verification and Execution of Mission Critical Applications, DASC'03.

[AW84] C. H. Applebaum, J. G. Williams. 1984. PVS - design for a practical verification system. ACM/CSC-Er, Proceedings of the 1984 annual conference of the ACM on The fifth generation challenge, pages: 58 - 68 .

[Bae88] R. Baecker. 1988. Enhancing program readability and comprehensibility with tools for program visualization. International Conference on Software Engineering Proceedings of the 10th international conference on Software engineering, pages: 356 – 366.

[BCM+92] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang. 1992. Symbolic Model Checking. Information and Computation, 98. pages: 142-170.

[BM88] R. Boyer, J. Moore. 1988. A Computational Logic. Academic Press Inc, London.

[BV97] T. Baker and T. Vardanega. 1997. Session summary: Tasking profiles. In A. Wellings, editor, ACM Ada Letters, *Proceedings of the 8th International Real-Time Ada Workshop*, pages: 5–7.

[BV99] S. Brown, Z. Vranesic. 1999. Fundamentals of Digital Logic with VHDL Design.

[BW01] A. Burns, A. Wellings. 2001. Real-Time Systems and Programming Languages (Third Edition). ISBN: 0201729881

[CDF+80] W. Carlson, L. Druffel, D. Fisher, W. Whitaker. 1980. Introducing Ada. Proceedings of the ACM 1980 annual conference, pages: 263-271.

[CDH+00] J. Corbett, J. Dwyer, J. Hatcliff, ... 2000. Bandera: extracting finite-state models from Java source code. International Conference on Software Engineering archive Proceedings of the 22nd international conference on Software engineering table of contents Limerick, Ireland, pages: 439 – 448.

[Cha98] P. Chapront. 1998. Ada + B the formula for safety critical software development. 1998. In L. Apslund (ed.), Reliable Software Technologies-Ada Europe, Vol. 1411 of Lecture Notes in Computer Science, pages: 14-18.

[CJ96] E. M. Clarke, J M. Wing. 1996. Formal methods: state of the art and future directions. ACM Computing Surveys (CSUR), Volume 28 Issue 4.

[CP96] C.T. Chou and D. Peled. 1996. Formal verification of a partial-order reduction technique for model checking. In T. Margaria and B. Steffen, editors, Tools and Algorithms for the Construction and Analysis of Systems, number 1055 in Lect. Notes Comp. Sci., Springer, Berlin, pages 241--257.

[DB98] B. Dobbing, and A. Burns. 1998. The Ravenscar tasking profile for high integrity real-time programs. In SIGAda '98.

[DI99] C. Demartini, R. Iosif, and R. Sisto. 1999.A deadlock detection tool for concurrent Java programs. Software – Practice and Experience, pages 577-603.

[DIS99] C. Demartini, R. Iosif, and R. Sisto. 1999. dSPIN: A dynamic extension of SPIN. In SPIN, pages: 261-276.

[DOT+96] C. Daws, A. Olivero, S. Tripakis and S. Yovine. 1996. The Tool KRONOS, in: Proceedings of Hybrid Systems III, LNCS 1066, pages: 208-219.

[Ead84] P. Eades. 1984. A Heuristic for Graph Drawing. Congr. Numer., 42. pages: 149–160.

[Elm90] P. Elmer-Dewitt. 1990. "Ghost in the Machine," Time Magazine, Jan. 29, 1990. page: 58.

[EM82] J. L. Elshoff, M. Marcotty. 1982. Improving computer program readability to aid modification. Communications of the ACM, pages: 512 – 521.

[FR91] T. Fruchterman and E. Reingold. 1991.Graph drawing by force-directed placement. *Softw. - Pract. Exp.*, 21(11):1129-1164.

[Gol94] D. Goldschlag. 1994. A Formal Model of Several VHDL concepts. Proceedings of the ninth annual conference on Computer Assurance, pages: 177-181.

[GRA04] http://www.ads.tuwien.ac.at/research/graphDrawing.html (May 14, 2004)

[Hal70] K. M. Hall. 1970. An r-dimensional Quadratic Placement Algorithm. Management Science 17, pages: 219-229.

[Hal86] Chris Hallgren. 1986. Factors affecting readability. ACM Special Interest Group for Design of Communications, Proceedings of the 4th annual international conference on Systems documentation, pages: 108 - 109.

[Han98] N. J. Haneef. 1998. Software documentation and readability: a proposed process improvement ACM SIGSOFT Software Engineering Notes archive Volume 23 , Issue 3, pages: 75 – 77.

[Har00] G. Hargis. 2000. Readability and computer documentation. ACM Journal of Computer Documentation Number 3, August 2000, pages: 122 – 131.

[Har87] D. Harel. 1987. Statecharts: A visualization formalism for complex systems. Science of Computer Programming 8. Elsevier Science Publishers B.V., North Holland, pages: 231-274.

[HD01] J. Hatcliff and M. Dwyer. 2001. Using the bandera tool set to model-check properties of concurrent java software. In International Conference on Concurrency Theory (CONCUR), Invited tutorial paper.

[HKL97] C. Heitmeyer, J. Kirby, B. Labaw. 1997. The SCR method for formally specifying, verifying, and validating requirements: tool support. International Conference on Software Engineering, Proceedings of the 19th international conference on Software engineering, Boston, Massachusetts, United States, pages: 610 – 611.

[HP99] K. Havelund and T. Pressburger. 1999. Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer.

[HU79] Hopcroft & Ullman. 1979. "Introduction to automata theory, languages and computations", Addison-Wesley.

[ICM93] Investigation Commission of Ministry of Transport – France. 1993. Rapport De la Commission d'Enquete sur l'Accident Survenu le 20 Janvier 1992 pres du Mont Saite Odile a l'Airbus A. 320 Imaarticule F-GGED Exploite par lay Compagnie Air Inter.

[KCH02] Y. Koren, L. Carmel, D. Harel. 2002. ACE: A Fast Multiscale Eigenvectors Computation for Drawing Huge Graphs.

[KG99] C. Kern, M. R. Greenstreet. 1999. Formal verification in hardware design: a survey. ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 4 , Issue 2, pages: 123 - 193.

[LHR99] N. Leveson, M. Heimdahl, J. Reese. 1999. Designing specification languages for process control systems: lessons learned and steps to the future. Foundations of Software Engineering, Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, pages: 127 – 145.

[LPY97] Kim G. Larsen, Paul Pettersson and Wang Yi. 1997. UPPAAL in a Nutshell. In Springer International Journal of Software Tools for Technology Transfer 1(1+2).

[MP98] M. Mills, G. Peterson. 1998. Hardware/Software Co-Design: VHDL and Ada 95 Code Migration and Integrated Analysis. SIGAda 1998. pages: 18-27.

[NBC+95] N. Njorner, A. Browne, E. Chang, M.Colon, A. Kapur, Z. Manna, H.B. Simpa, T.E. Uribe. 1995. Step: The Stanford Temporal Prover- user's manual. Technical Report STAN-CS_TR-95-1562, Department of Computer Science, Stanford University.

[Niv94] F. Nivoli. 1994. Formal verification of behavioral VHDL specifications: a case study. Proceedings of the conference on European design automation, pages: 560 - 565.

[PL00] Paul Pettersson and Kim G. Larsen. 2000. Uppaal2k. Bulletin of the European association for theoretical computer science, volume 70, pages 40-44.

[RSK98] R. Reetz, K. Schneider, T. Kropf. 1998. Formal specification in VHDL for hardware verification Design, Automation, and Test in Europe proceedings of the conference on Design, automation and test in Europe, pages: 257 - 265

[RW98] W. Reisig, Wolfgang. 1998. Elements of Distributed Algorithms – Modelling and Analysis with Petri Nets. Springer, 1998.

[Sch01] J. M. Schumann. 2001. automated theorem proving in software engineering. Springer Verlag, 2001, xiv+228 pages, ISBN 3-540-67989-8

[SG02] S. Katz, O. Grumberg. 2002. A framework for Translating Models and Specifications. Proceedings of IFM2002 (International Conference on Integrated Formal Methods), LNCS 2335, May 2002, pages: 145-164.

[Sha86] M. Shahdad. 1986. An overview of VHDL language and technology . Proceedings of the 23rd ACM/IEEE conference on Design automation.

[Sil04] A. Silbovitz. 2004. The Ravenscar-compliant hardware run-time (RavenHaRT) kernel. Massachusetts Institute of Technology.

[Sim96] S. Sim. Automatic Graph Drawing Algorithms. simsuz@turing.utoronto.ca. December 17, 1996.

[Sip97] M. Sipser. 1997. Introduction to the Theory of Computation. Published by the PWS Publishing Company. ISBN: 053494728X

[SL96] H. Sogame, P. Ladkin. 1996. "Nagoya A300 - Aircraft Accident Investigation Report 96-5,". http://www.techfak.uni-bielefeld.de/~ladkin/nagoyarep/nagoya-top.html (23 April, 2004)

[Upp04] http://www.uppaal.com. (May 4,2004)

[VKB+97] T. Villa, T. Kam, R. Brayton, A. Sangiovanni-Vincentelli. 1997. Synthesis of Finite State Machines: Functional Optimization, Kluwer Academic Publishers, ISBN 0792398424

[VSP] VSPEC web page at http://mint.cs.man.ac.uk/Projects/UPC/Languages/Larch.html

[ZLL02] M. Zimmerman, K. Lundqvist, N. Leveson, 2002, Investigating the Readability of State-Based Formal Requirements Specification Languages, International Conference on Software Engineering, ICSE'02, pages: 33-43.

8. Appendix A

This appendix contains the definition files (.ads) for the different packages comprising the VAT tool. There are nine definition files:

- Conditional_List_Handlers.ads
- Output_Functions.ads
- Parser_Types.ads
- Statement_Handlers.ads
- Table_Setup_Function.ads
- Table_Write_Functions.ads
- Truth_Table_Grapher.ads
- Graphing4.ads

The pre and post conditions are provided for the different functions and procedures.

```
CONDITIONAL_LIST_HANDLERS.ADS
-------------------------------------------------------------------
-----------------------------------------------
package conditional_list_Handlers is
```

FUNCTION Type_Of_Conditional
Pre: A pointer to a conditional node.
Post: Returns type of conditional node.

```
function Type_Of_Conditional (
Word : in     Wide_String_Ptr;
Signal_List : Signal_List_Node_Ptr;
Input_List_Head_Ptr: Input_List_Node_Ptr;
Output_List_Head_Ptr: Output_List_Node_Ptr )
return Conditional_Type;
```

PROCEDURE Copy_Conditional_List
Pre: A list of type Conditional_Node_Ptr.
Post: Returns a copy of the list.

```
procedure Copy_Conditional_List (
List1      : in     Conditional_Node_Ptr;
List2 :    out Conditional_Node_Ptr  );
```

PROCEDURE Add_Conditional_Lists
Pre: Two lists of type Conditional_Node_Ptr.
Post: Returns the second list as the concatenation of the two lists.

```
procedure Add_Conditional_Lists(
Conditional_List: in out Conditional_Node_Ptr;
New_List: in Conditional_Node_Ptr);
```

FUNCTION Just_Negation
Pre: A conditional list.
Post: Returns true if the conditional list if a negation list.

```
function just_negation(
Conditional_List:  Conditional_Node_Ptr) return boolean;
```

FUNCTION Add_Element_To_Conditional_List
Pre: A conditional list and a conditional node.
Post: Returns new list with node attached to original list.

```
function Add_Element_To_Conditional_List (
Conditional_List      :   Conditional_Node_Ptr;
type_of_conditional_variable : Conditional_Node_Type;
Name : Wide_String_Ptr;
Operator:Wide_String_Ptr;
value : Wide_String_Ptr) return Conditional_Node_Ptr;
```

PROCEDURE Print_Conditional_List
Pre: A conditional list.
Post: Prints contents of list to console.

```
procedure Print_Conditional_List (
List1 :  Conditional_Node_Ptr);
```

PROCEDURE Append_Element_To_Conditional_List
Pre: A conditional list and a conditional node.
Post: Adds node to list and returns updated list.

```
procedure Append_Element_To_Conditional_List (
List1      :  in out Conditional_Node_Ptr;
type_of_conditional_variable : Conditional_Node_Type;
Name : Wide_String_Ptr;
Operator:Wide_String_Ptr;
value : Wide_String_Ptr);
```

```
end conditional_list_Handlers;
-------------------------------------------------------------
-------------------------------------------------
```

```
OUTPUT_FUNCTIONS.ADS
-------------------------------------------------------------
-------------------------------------------------
package Output_Functions is
```

PROCEDURE Print_Signal_Names
Pre: Signal list must be complete (signals in file must have been collected).
Post: Lists all signal names onto output (ex: screen).

```
procedure Print_Signal_Names(Signal_List : Signal_List_Node_Ptr);
```

PROCEDURE Print_Input_Names
Pre: Input variable list must be complete (input variables in file must have been collected).
Post: Lists all input variable names onto output (ex: screen).

```
procedure Print_Input_Names(Input_List : Input_List_Node_Ptr);
```

PROCEDURE Print_Output_Names
Pre: Output variable list must be complete (output variables in file must have been collected).
Post: Lists all output variable names onto output (ex: screen).

```
procedure Print_Output_Names(Output_List : Output_List_Node_Ptr);
```

```
--------------------------------------------------------------------------------
```

PROCEDURE Print_Table_Info
Pre: All pointers are stored in files.
Post: Lists all pointers in a single table onto output (ex: screen).

```
procedure Print_Table_Info(State_Variable_List: State_variable_ptr);
--------------------------------------------------------------------------------


end Output_Functions;
--------------------------------------------------------------------------------
--------------------------------------------------


PARSER_TYPES.ADS
--------------------------------------------------------------------------------
--------------------------------------------------
package parser_types is

      MAXIMUM_STRING_LENGTH : CONSTANT INTEGER := 200;
      subtype Lines is String(1..MAXIMUM_STRING_LENGTH);
      Tab:   Character  := Character'Val (9);


   type wide_string_ptr is access wide_String;

   ------------------------------------------------------------------
   type Conditional_Type is
         (Input_Variable,
          Output_Variable,
          State_Variable,
          NONE);

   ------------------------------------------------------------------


   ------------------------------------------------------------------
   type Statement_Type is
      (If_Statement,
      Case_Statement,
      Output_Assignment_Statement,
      State_Assignment_Statement,
      Signal_Assignment_Statement,
      Unknown_Statement
         );
   ------------------------------------------------------------------

   ------------------------------------------------------------------
   type Conditional_Node_Type is
      (Input_Conditional,
      Signal_Conditional);

   ------------------------------------------------------------------
```

```
-----------------------------------------------------------------
      ■ THIS IS USED AS A LIST OF THE INPUT VALUES THAT CAUSE A
        CERTAIN TRANSITION OR OUTPUT VALUE
-----------------------------------------------------------------
type Input_Value_Node;
type Input_Value_Node_Ptr is access Input_Value_Node;
type Input_Value_Node is
   record
      Input_Name  : wide_string_ptr;
      Input_Value : Integer;
      Next        : Input_Value_Node_Ptr;
   end record;
-----------------------------------------------------------------

-----------------------------------------------------------------
   ■ THIS IS USED AS A LIST OF THE SIGNAL VALUES THAT CAUSE A
     CERTAIN TRANSITION OR OUTPUT VALUE
-----------------------------------------------------------------
type Signal_Name_Value_Node;
type Signal_name_Value_Node_Ptr is access Signal_Name_Value_Node;
type Signal_name_Value_Node is
   record
      Signal_Name  : wide_string_ptr;
      Signal_Value : wide_string_ptr;
      priority     : integer;
      Next         : Signal_Name_Value_Node_Ptr;
   end record;
-----------------------------------------------------------------


-----------------------------------------------------------------
   ■ THIS IS USED AS A LIST OF THE INPUT VALUES AND SIGNAL
     VALUES THAT CAUSE A CERTAIN TRANSITION OR OUTPUT VALUE
-----------------------------------------------------------------
type Conditional_Node;
type Conditional_Node_Ptr is access Conditional_Node;
type Conditional_Node is
record
      Type_Of_Node : Conditional_Node_Type;
      Name : Wide_String_Ptr;
      Operator: Wide_String_Ptr;
      Value : wide_string_ptr;
      Next         : Conditional_Node_Ptr;
   end record;
-----------------------------------------------------------------


-----------------------------------------------------------------
   ■ THIS IS USED AS A LIST OF ALL THE INPUT VARIABLES IN THE
     AUTOMATA
-----------------------------------------------------------------
type Input_List_Node;
type Input_List_Node_Ptr is access Input_List_Node;
type Input_List_Node is
   record
```

```
      Input_Name : wide_string_ptr;
      Next       : Input_List_Node_Ptr;
   end record;
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
   ■ THIS IS USED AS A LIST OF ALL THE OUPUT VARIABLES IN THE
     AUTOMATA
-----------------------------------------------------------------
type Output_List_Node;
type Output_List_Node_Ptr is access Output_List_Node;
type Output_List_Node is
   record
      Output_Name : wide_string_ptr;
      Next        : Output_List_Node_Ptr;
   end record;
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
   ■ THIS IS USED AS A LIST OF ALL THE SIGNALS IN THE AUTOMATA
-----------------------------------------------------------------
type Signal_List_Node;
type Signal_List_Node_Ptr is access Signal_List_Node;
type Signal_List_Node is
   record
      Signal_Name : wide_string_ptr;
      Next        : Signal_List_Node_Ptr;
   end record;
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
   ■ This represents the value of an output variable
     corresponding to a combination of input values
-----------------------------------------------------------------
type Output_Value_Node;
type Output_Value_Node_Ptr is access Output_Value_Node;
type Output_Value_Node is
   record
      Input_Combinations : Conditional_Node_Ptr;
      Output_Value       : WIde_String_Ptr;
      Next               : Output_Value_Node_Ptr;
   end record;
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
   ■ Each output variable has one of these which represents an
     output column in a truth table
-----------------------------------------------------------------
type Output_Column;
type Output_Column_Ptr is access Output_Column;
type Output_Column is
   record
      Output_Name               : wide_string_ptr;
      Output_Value_List_Head_Ptr : Output_Value_Node_Ptr;
```

```
      Next                          : Output_Column_Ptr;
   end record;
-------------------------------------------------------------------

-------------------------------------------------------------------
type Signal_Value_Node;
type Signal_Value_Node_Ptr is access Signal_Value_Node;
type Signal_Value_Node is
   record
     Input_Combinations : Conditional_Node_Ptr;
     Signal_Value       : WIde_String_Ptr;
     Next               : Signal_Value_Node_Ptr;
   end record;
-------------------------------------------------------------------

-------------------------------------------------------------------
type Signal_Column;
type Signal_Column_Ptr is access Signal_Column;
type Signal_Column is
   record
      Signal_Name                : wide_string_ptr;
      Signal_Value_List_Head_Ptr : Signal_Value_Node_Ptr;
      Next                       : Signal_Column_Ptr;
   end record;
-------------------------------------------------------------------


-------------------------------------------------------------------
   ■ Each next state variable has one of these which represents a
     column in a truth table
-------------------------------------------------------------------
type Next_State_Node;
type Next_State_Node_Ptr is access Next_State_Node;
type Next_State_Node is
   record
      Conditions : Conditional_Node_Ptr;
      Next_State_Name    : wide_string_ptr;
      priority           : integer;
      Next               : Next_State_Node_Ptr;
   end record;
-------------------------------------------------------------------


-------------------------------------------------------------------
type Row;
type Row_Ptr is access Row;
type Row is
   record
      Present_State          : wide_string_ptr;
      Next_State_Head_Ptr    : Next_State_Node_Ptr;
      Output_Column_List_Ptr : Output_Column_Ptr;
      Signal_Column_List_ptr : Signal_Column_Ptr;
      Location_X             : Integer;
      Location_Y             : Integer;
      Next                   : Row_Ptr;
      Global_Priority_Counter : integer := 0;
   end record;
```

```
   ------------------------------------------------------------------

   ------------------------------------------------------------------
   type State_Variable_Node;
   type State_Variable_Ptr is access State_Variable_Node;
   type State_Variable_Node is
      record
         Signal_Name    : Wide_String_Ptr;
         Start_State    : Wide_String_Ptr;
         Truth_Table     : Row_Ptr;
         Next            : State_Variable_Ptr;
      end record;
   ------------------------------------------------------------------

   ------------------------------------------------------------------
   type Signal_Equivalency_List;
   type Signal_Equivalency_List_Ptr is access
        Signal_Equivalency_List;
   type Signal_Equivalency_List is
      record
         Signal_Name : Wide_String_Ptr;
         Equivalent_Signal : WIde_String_Ptr;
         Next        : Signal_Equivalency_List_Ptr;
      end record;
   ------------------------------------------------------------------



   State_variable_list  : State_variable_Ptr   := null;
   State_Equivalency_list : Signal_Equivalency_List_Ptr := null;

end parser_types;
-----------------------------------------------------------------
-------------------------------------------------

PARSING_FUNCTIONS.ADS
-----------------------------------------------------------------
-------------------------------------------------
package Parsing_Functions is
```

PROCEDURE Get_Next_Word

Pre: Located at some point in input file.

Post: All delimiters, spaces, tabs, parenthesis, semicolons, colons, commas, equal signs, and less than and greater than signs, have been ignored. String following those characters has been extracted. Cursor is exactly after last non-delimiter character.

```
procedure Get_Next_Word (Input: in Ada.Text_Io.File_Type; Delimiter: in
Character; Word: out Lines; Length: out Integer; Eof: out Boolean);
```

FUNCTION Get_Next_Operator
Pre: Located at some point in input file.
Post: Extracted all equal signs, and less than and greater than signs. All other characters are ignored.

```
function Get_Next_Operator (Input  :  in    Ada.Text_Io.File_Type)
return Wide_String_Ptr;
```

FUNCTION Next_Word
Pre: Located at some point in input file.
Post: All delimiters, spaces, tabs, parenthesis, semicolons, colons, commas, equal signs, and less than and greater than signs, are delimiters. Strings following or preceding those characters ae extracted. Cursor is exactly after last non-delimiter character.

```
function Next_Word (Input:  in Ada.Text_Io.File_Type; Delimiter: in
Character) return Wide_String_Ptr;
```

PROCEDURE Special_Get_Next_Word
Pre: Located at some point in input file.
Post: String formed by all characters after first delimiter excluding commas, spaces, tabs.

```
procedure Special_Get_Next_Word (Input: in Ada.Text_Io.File_Type;
Delimiter: in Character; Word: out Lines; Length: out Integer; Eof: out
Boolean);
```

PROCEDURE Find_Word
Pre: Located at beginning of input file.
Post: Identifies first occurrence of word in file. Cursor is exactly before first character of word.

```
procedure Find_Word (Input  : in out Ada.Text_Io.File_Type;
Search_Word : in String; eof: out boolean);
```

FUNCTION Find_Next_Matching_Word
Pre: Located at some point in input file.
Post: Identifies next matching word in file. Cursor is exactly before first character of matching word.

```
function Find_Next_Matching_Word (Input  :  in    Ada.Text_Io.File_Type;
Search_Word : in String ) return integer;
```

PROCEDURE Find_Next_Conditional_Adjective
Pre: Located at some point in input file.
Post: Finds next conditional adjective. Cursor is exactly before first character of conditional adjective.

```
procedure Find_Next_Conditional_Adjective (Input: in
Ada.Text_Io.File_Type; Word: out Lines; Word_Length: out Integer; Eof:
out Boolean);
```

PROCEDURE Find_Next_Variable_Or_Then
Pre: Located at some point in input file.
Post: Found next variable or conditional statement "then". Cursor is exactly before first character of word found.

```
procedure Find_Next_Variable_Or_Then (Input: in Ada.Text_Io.File_Type;
Input_List_Head_Ptr : in Input_List_Node_Ptr; Signal_List : in
Signal_List_Node_Ptr; Word: out Lines; Word_Length: out Integer; Eof:
out Boolean; type_of_conditional_variable : out Conditional_Node_Type);
```

PROCEDURE Find_Next_State_Name
Pre: Located at some point in input file.
Post: Finds next state name. Cursor is exactly before first character of state name.

```
procedure Find_Next_State_Name (Input: in Ada.Text_Io.File_Type; Word:
out Lines; Word_Length: out Integer; Eof: out Boolean; Row_Head:
Row_Ptr);
```

PROCEDURE Find_Next_Matching_Character
Pre: Located at some point in input file.
Post: Cursor is exactly after last matching character retrieved.

```
procedure Find_NExt_matching_Character (
         Input  :  in    Ada.Text_Io.File_Type;
         Char: in Character  );
```

FUNCTION Compare_Words
Pre: Located at some point in input file.
Post : Compared two words and returned true if equal, false if not.

```
function Compare_Words(
      String1 : String;
```

```
      String2 : String) return Boolean;
```

PROCEDURE Get_Until
Pre: Located at some point in input file.
Post: All delimiters, apostrophes, tabs, parenthesis, equal signs, less than and greater than signs, spaces, and concatenation signs are ignored. String following those characters is extracted. Cursor is exactly after last non-delimiter character.

```
procedure Get_until (Input  :  in    Ada.Text_Io.File_Type;Delimiter:
in Character;Word:out Lines;Length:out Integer;Eof:out Boolean    );
```

PROCEDURE Find_End_For
Pre: Located at some point in input file.
Post: Finds end of input file. Cursor is exactly after last character of input file.

```
procedure Find_End_For(
      Input  :  in    Ada.Text_Io.File_Type;
      String1 : String);
```

```
end Parsing_Functions;
```

```
STATEMENT_HANDLERS.ADS
```

```
package Statement_Handlers is
```

PROCEDURE Process_If_Statement
Pre: Located at after the IF construct.
Post: Processes the IF statement.

```
procedure Process_If_Statement (
     Input  :  in    Ada.Text_Io.File_Type;
     State_Info: Signal_Name_Value_Node_Ptr;
     Conditional_List: Conditional_Node_Ptr;
     Input_List_Head_Ptr: Input_List_Node_Ptr;
     Signal_List : Signal_List_Node_Ptr;
     Output_List_Head_Ptr: Output_List_Node_Ptr);
```

PROCEDURE Process_State_Assignment_Statement
Pre: Located at beginning of state assignment statement.
Post: Processes the state assignement statement.

```
procedure Process_State_Assignment_Statement(
      Input  :  in    Ada.Text_Io.File_Type;
      State_Info: Signal_Name_Value_Node_Ptr;
      Conditional_List: in Conditional_Node_Ptr;
      First_Word: Wide_String_Ptr);
```

PROCEDURE Process_Output_Assignment_Statement
Pre: Located at beginning of output assignment statement.
Post: Processes the output assignement statement.

```
procedure Process_Output_Assignment_Statement(
      Input  :  in    Ada.Text_Io.File_Type;
      State_Info: Signal_Name_Value_Node_Ptr;
      Conditional_List: in Conditional_Node_Ptr;
      First_Word: Wide_String_Ptr);
```

PROCEDURE Process_Signal_Assignment_Statement
Pre: Located at beginning of signal assignment statement.
Post: Processes the signal assignement statement.

```
procedure Process_Signal_Assignment_Statement(
      Input  :  in    Ada.Text_Io.File_Type;
      State_Info: Signal_Name_Value_Node_Ptr;
      Conditional_List: in Conditional_Node_Ptr;
      First_Word: Wide_String_Ptr);
```

FUNCTION Process_Statement
Pre: Located at beginning of any statement.
Post: Function finds out the nature of the statement and makes a call to one of the specialized handling functions.

```
function Process_Statement(
      Input  :  in    Ada.Text_Io.File_Type;
      State_Info: Signal_Name_Value_Node_Ptr;
      Conditional_List:  Conditional_Node_Ptr;
      Input_List_Head_Ptr: Input_List_Node_Ptr;
      Signal_List : Signal_List_Node_Ptr;
      Output_List_Head_Ptr: Output_List_Node_Ptr ) return integer;
```

PROCEDURE Process_Case _Statement
Pre: Located at beginning of CASE statement, right after the CASE.
Post: Processes the WHEN clauses of the CASE statement.

```
procedure Process_Case_Statement(
      Input  :  in    Ada.Text_Io.File_Type;
      State_Info: Signal_Name_Value_Node_Ptr;
      Conditional_List: Conditional_Node_Ptr;
      State_Variable_Name: out Wide_String_Ptr;
      Signal_List : Signal_List_Node_Ptr;
      Input_List_Head_Ptr: Input_List_Node_Ptr;
      Output_List_Head_Ptr: Output_List_Node_Ptr );
```

FUNCTION Type_Of_Statement
Pre: Takes a token as an input.
Post: Function finds out the type of statement the token represents and returns the type.

```
function Type_Of_Statement(
      State_Info: Signal_Name_Value_Node_Ptr;
      First_Word : in     Wide_String_Ptr;
      Signal_List : Signal_List_Node_Ptr;
      Input_List_Head_Ptr: Input_List_Node_Ptr;
      Output_List_Head_Ptr: Output_List_Node_Ptr   )  return
      Statement_Type;
```

```
end Statement_Handlers;
-------------------------------------------------------------------
-------------------------------------------------
```

```
TABLE_SETUP_FUNCTIONS.ADS
-------------------------------------------------------------------
-------------------------------------------------
package Table_Setup_Functions is
```

PROCEDURE Input_Output_Variables
Pre: Located at beginning of input file.
Post: Creates signal list containing names of all signal variables.

```
procedure Input_Output_Variables (Input  : in out
Ada.Text_Io.File_Type; Input_List_Head_Ptr : in out Input_List_Node_Ptr;
Output_List_Head_Ptr : out Output_List_Node_Ptr);
```

PROCEDURE Signals
Pre: Located at beginning of input file.
Post: Creates signal list containing names of all signal variables.

```
procedure Signals (Input  : in out   Ada.Text_Io.File_Type;
Signal_List : out Signal_List_Node_Ptr );
```

PROCEDURE Create_Output_Column
Pre: Output column does not exist.
Post: Creates output column for a particular truth table.

```
procedure Create_Output_Column (Output_List_Head_Ptr : in
Output_List_Node_Ptr; Output_Column_List_Ptr : out Output_Column_Ptr);
```

PROCEDURE Create_Signal_Column
Pre: Signals column does not exist.
Post: Creates signals column for a particular truth table.

```
procedure Create_Signal_Column (signal_list_Head_ptr : in
signal_List_Node_Ptr; signal_Column_List_Ptr : out signal_Column_Ptr);
```

PROCEDURE Add_Row
Pre: No Row for particular state.
Post: Adds a row for a new state in a particular truth table.

```
   procedure Add_Row (
      Output_List_Head_Ptr : Output_List_Node_Ptr;
      signal_List_Head_Ptr : signal_List_Node_Ptr;
      State_Variable :  Wide_String_Ptr;
      State_Name : Wide_String_Ptr;
      State_Variable_List : in out State_Variable_Ptr);
```

PROCEDURE Create_State_Variable_Entry
Pre: State variable is needed.
Post: Creates state variable entry.

```
procedure Create_State_Variable_Entry(State_Variable_list : in out
State_Variable_ptr; Signal_Name : Wide_String_ptr);
```

```
end table_Setup_functions;
```

```
-------------------------------------------------------------------------
-----------------------------------------------
```

```
TABLE_WRITE_FUNCTIONS.ADS
-------------------------------------------------------------------------
-----------------------------------------------
package Table_Write_Functions is
```

PROCEDURE Fill_In_Transition
Pre: Transition not entered
Post: Enters a transition in the next state column of the truth table

```
procedure Fill_In_Transition (
      State_Info: Signal_Name_Value_Node_Ptr;
      Next_State : in     Wide_String_Ptr;
      Conditions_List :    Conditional_Node_Ptr;
      State_Variable_list: State_Variable_ptr );
```

PROCEDURE Fill_In_Output_Value
Pre: Output Assignment function not entered
Post: Enters an output assignment function into truth table

```
procedure Fill_In_Output_Value (
      State_Info: Signal_Name_Value_Node_Ptr;
      Output_Name    : in     Wide_String_Ptr;
      Output_Value : in Wide_String_Ptr;
      Conditions_List :    Conditional_Node_Ptr;
      State_Variable_list: State_Variable_ptr);
```

PROCEDURE Fill_In_Signal_Value
Pre: Signal Assignment function not entered
Post: Enters a signal assignment function into truth table

```
procedure Fill_In_Signal_Value (
      State_Info: Signal_Name_Value_Node_Ptr;
      Signal_Name    : in     Wide_String_Ptr;
      Signal_Value : in Wide_String_Ptr;
      Conditions_List :    Conditional_Node_Ptr;
      State_Variable_list: State_Variable_ptr );
```

```
end table_Write_functions;
-------------------------------------------------------------------------
-----------------------------------------------
```

TRUTH_TABLE_GRAPHER.ADS

```
-------------------------------------------------------------------------------
-----------------------------------------------
package Truth_Table_Grapher is

-------------------------------------------------------------------------------
```

PROCEDURE Grapher
Pre: There is at least one truth table.
Post: All truth tables are graphed.

```
procedure Grapher (State_list : in out State_variable_ptr);
-------------------------------------------------------------------------------

end Truth_Table_Grapher;
-------------------------------------------------------------------------------
-----------------------------------------------
```

GRAPHING4.ADS

```
-------------------------------------------------------------------------------
-----------------------------------------------
package graphing4 is


   Tab:   Character := Character'Val (9);
-----------------------------------------------------------
-------------------------------------------------------------------
type Node is
   record
      X : double;
      Y : double;
      Dx : double;
      Dy : double;
      Initial : Boolean := False;
      Urgent : Boolean := false;
      Committed : Boolean := false;
      Name : Wide_String_Ptr;
      Invariant : Wide_String_Ptr;
   end record;
-------------------------------------------------------------------

-------------------------------------------------------------------
-------------------------------------------------------------------
type Edge is
   record
      from : integer;
      to : integer;
      len : double;
   end record;
-------------------------------------------------------------------


Nnodes :Integer := 1;
Nedges :Integer := 1;
```

```
type Node_Array is array(Integer range <>) of Node;
Nodes : Node_Array(1..300);
type Edge_Array is array(Integer range <>) of Edge;
Edges : Edge_Array(1..300);
```

FUNCTION FtoS
Pre: Takes a float as an input.
Post: Function returns a String representation of the float.

```
function FtoS (F : Float ) return String;
```

FUNCTION Addnode
Pre: Takes state information as an input.
Post: Adds a corresponding node representing the state.

```
function Addnode (Name : in   Wide_String_Ptr; Invariant :
   wide_string_ptr:= null; Urgent :
   boolean := false; Committed : boolean := false; Initial :
   boolean := false) return Integer;
```

FUNCTION Findnode
Pre: Takes state information as an input.
Post: Return corresponding node ID.

```
function Findnode (Name : in   Wide_String_Ptr) return Integer;
```

PROCEDURE addedge
Pre: Takes in id's for two nodes and a length Len.
Post: Adds an edge between them with length Len.

```
procedure addEdge (from : in   Wide_String_Ptr;
                   to : in Wide_String_Ptr;
                   Len : Double
                   );
```

PROCEDURE Relax
Pre: Takes in a seed for a random generator. Graph info must exist in data structures.
Post: Applies the modified Spring Embdder algorithm to the graph resident in the data structures.

```
procedure Relax (Seed : in
```

```
   Ada.Numerics.Float_Random.Generator);
```

PROCEDURE Find_location
Pre: Takes in the name of a state and two integers X and Y.
Post: Stores coordinates of node representing the state in the X and Y integers.

```
procedure Find_location(name : wide_string_ptr; X : out
   integer; Y : out integer);
   Seed : Ada.Numerics.Float_Random.Generator;
```

```
end graphing4;
```

9. Appendix B

Appendix B contains the code (.adb) for the different packages comprising the VAT tool. There are nine files in total:

- Conditional_List_Handlers.adb
- Output_Functions.adb
- Statement_Handlers.adb
- Table_Setup_Function.adb
- Table_Write_Functions.adb
- Truth_Table_Grapher.adb
- Graphing4.adb
- Parser_Advanced6.adb

```
CONDITIONAL_LIST_HANDLERS.ADB
-------------------------------------------------------------------------
-----------------------------------------------

package body Conditional_List_Handlers is


   ---------------------------------------------------------------
   function Type_Of_Conditional (
         Word : in     Wide_String_Ptr;
         Signal_List : Signal_List_Node_Ptr;
         Input_List_Head_Ptr: Input_List_Node_Ptr;
         Output_List_Head_Ptr: Output_List_Node_Ptr ) return
   Conditional_Type is

      Ptr1 : Input_List_Node_Ptr;
      Ptr2 : Output_List_Node_Ptr;
      Ptr3 : Signal_List_Node_Ptr;
```

```
begin

   Ptr3:= Signal_List;
   loop
      exit when Ptr3 = null;
      if
      (Compare_Words(Wide_String_To_String(Ptr3.Signal_Name.all,1),
      Wide_String_To_String(Word.all,1))) then
         return State_Variable;
      end if;
      Ptr3 := Ptr3.Next;
   end loop;

   Ptr1:= Input_List_Head_Ptr;

   loop
      exit when Ptr1 = null;
      if  (Compare_Words(Wide_String_To_String(Ptr1.Input_Name.all,1),
          Wide_String_To_String(Word.all,1))) then
         return Input_Variable;
      end if;
      Ptr1 := Ptr1.Next;
   end loop;

   Ptr2:= Output_List_Head_Ptr;
   loop
      exit when Ptr2 = null;
      if(Compare_Words(Wide_String_To_String(Ptr2.Output_Name.all,1),
      Wide_String_To_String(Word.all,1))) then
         return Output_Variable;
      end if;
      Ptr2 := Ptr2.Next;
   end loop;

   return NONE;

end Type_Of_Conditional;
-------------------------------------------------------------------


-------------------------------------------------------------------
procedure Copy_Conditional_List (
      List1     : in     Conditional_Node_Ptr;
      List2 :    out Conditional_Node_Ptr  ) is

   Temp  : Conditional_Node_Ptr;
   Temp1 : Conditional_Node_Ptr;
   Temp2_TAIL : Conditional_Node_Ptr;

begin

   if (List1 = null) then
      List2 := null;
   else
      Temp  := List1;
```

```
      Temp1 := new Conditional_Node;
      Temp1.Name := Temp.Name;
      Temp1.Operator := Temp.Operator;
      Temp1.Value := Temp.Value;
      Temp1.Next := null;
      List2 := Temp1;
      Temp2_TAIL := Temp1;

      loop
         Temp := Temp.Next;
         exit when Temp = null;
         Temp1 := new Conditional_Node;
         Temp1.Name := Temp.Name;
         Temp1.Operator := Temp.Operator;
         Temp1.Value := Temp.Value;
         Temp1.Next := null;
         Temp2_Tail.Next := Temp1;
         Temp2_Tail := Temp1;
      end loop;
   end if;

end Copy_Conditional_List;
-------------------------------------------------------------------


-------------------------------------------------------------------
procedure Add_Conditional_Lists(
      Conditional_List: in out Conditional_Node_Ptr;
      New_List: in Conditional_Node_Ptr) is

   Temp_Ptr : Conditional_Node_Ptr;

begin

   if (Conditional_List = null) then
      Conditional_List := New_List;
   else
      Temp_Ptr := conditional_List;
      loop
         exit when Temp_Ptr.next = null;
         Temp_Ptr := Temp_Ptr.Next;
      end loop;
      Temp_Ptr.Next := new_List;
   end if;

end Add_Conditional_Lists;
-------------------------------------------------------------------


-------------------------------------------------------------------
function Just_Negation(
      Conditional_List:  Conditional_Node_Ptr) return boolean
is

   Temp_Ptr : Conditional_Node_Ptr;
```

```
begin

      Temp_Ptr := conditional_List;
      loop
         exit when Temp_Ptr = null;
         if (Temp_Ptr.Name.All(1) /= '*') then
            return false;
         end if;
         Temp_Ptr := Temp_Ptr.Next;
      end loop;
      return true;
end Just_Negation;
-------------------------------------------------------------------


-------------------------------------------------------------------
function Add_Element_To_Conditional_List (
   Conditional_List     :   Conditional_Node_Ptr;
   type_of_conditional_variable : Conditional_Node_Type;
   Name : Wide_String_Ptr;
   Operator:Wide_String_Ptr;
   value : Wide_String_Ptr) return Conditional_Node_Ptr is

   Temp_Ptr  : Conditional_Node_Ptr;
   Temp1 : Conditional_Node_Ptr;
   List1     :   Conditional_Node_Ptr;

begin

   Copy_Conditional_List(Conditional_List,List1);

   Temp_Ptr := new Conditional_Node;
   Temp_Ptr.Type_Of_Node := type_of_conditional_Variable;
   Temp_Ptr.Name := Name;
   Temp_Ptr.Operator := Operator;
   Temp_Ptr.Value := Value;
   Temp_Ptr.Next := null;

   if (List1 = null) then
      List1 := Temp_Ptr;
   else
      Temp1  := List1;
      loop
         exit when Temp1.next = null;
         Temp1 := Temp1.next;
      end loop;
      Temp1.next := Temp_Ptr;
   end if;

   return List1;

end Add_Element_To_Conditional_List;
-------------------------------------------------------------------
```

```
-------------------------------------------------------------------------
procedure Print_Conditional_List (
   List1      :  Conditional_Node_Ptr)  is

   Temp_Ptr  : Conditional_Node_Ptr;
   Temp1 : Conditional_Node_Ptr;

begin

   if (List1 = null) then
      Ada.Text_IO.Put_Line("EMPTY LIST!");
   else
      Ada.Text_IO.Put_Line("LIST CONTENTS");
      Temp1  := List1;
      loop
         exit when Temp1 = null;
          Ada.Text_IO.Put_Line
          (Wide_String_To_String(Temp1.name.all,1));
         Temp1 := Temp1.next;
      end loop;
      Ada.Text_IO.Put_Line("-----------------");
   end if;


end Print_Conditional_List;
-------------------------------------------------------------------------


-------------------------------------------------------------------------
procedure Append_Element_To_Conditional_List (
   List1      :  in out Conditional_Node_Ptr;
   type_of_conditional_variable : Conditional_Node_Type;
   Name : Wide_String_Ptr;
   Operator : Wide_String_Ptr;
   value : Wide_String_Ptr)  is

   Temp_Ptr  : Conditional_Node_Ptr;
   Temp1 : Conditional_Node_Ptr;

begin

   Temp_Ptr := new Conditional_Node;
   Temp_Ptr.Type_Of_Node := type_of_conditional_Variable;
   Temp_Ptr.Name := Name;
   Temp_Ptr.Operator := Operator;
   Temp_Ptr.Value := Value;
   Temp_Ptr.Next := null;

   if (List1 = null) then
      List1 := Temp_Ptr;
   else
      Temp1  := List1;
      loop
         exit when Temp1.next = null;
         Temp1 := Temp1.next;
      end loop;
      Temp1.next := Temp_Ptr;
```

```
      end if;

   end Append_Element_To_Conditional_List;
   ------------------------------------------------------------

end conditional_list_handlers;
-------------------------------------------------------------------
-------------------------------------------------



OUTPUT_FUNCTIONS.ADB
-------------------------------------------------------------------

-------------------------------------------------

package body output_functions is


   ------------------------------------------------------------
   procedure Print_Signal_Names(Signal_List :Signal_List_Node_Ptr)is
      Temp7          : Signal_List_Node_Ptr;

   begin

      Ada.Text_Io.Put_Line(" ");
      Ada.Text_Io.Put_Line("-----------------------------");
      ada.Text_IO.Put_Line("SIGNALS IN FILE:");

      Temp7 := Signal_List;
      loop
         exit when Temp7 = null;
         Put_Line(Wide_String_To_String(Temp7.Signal_Name.all,1));
         Temp7 := Temp7.Next;
      end loop;

      Ada.Text_Io.Put_Line("-----------------------------");

   end Print_Signal_Names;
   ------------------------------------------------------------


   ------------------------------------------------------------
   procedure Print_Input_Names(Input_List : Input_List_Node_Ptr)is
      Temp7          : Input_List_Node_Ptr;

   begin

      Ada.Text_Io.Put_Line(" ");
      Ada.Text_Io.Put_Line("-----------------------------");
      ada.Text_IO.Put_Line("INPUTS IN FILE:");

      Temp7 := Input_List;
```

```
    loop
       exit when Temp7 = null;
       Put_Line(Wide_String_To_String(Temp7.Input_Name.all,1));
       Temp7 := Temp7.Next;
    end loop;

    Ada.Text_Io.Put_Line("-----------------------------");

end Print_Input_Names;
-------------------------------------------------------------------


-------------------------------------------------------------------
procedure Print_Output_Names(Output_List :Output_List_Node_Ptr)is
    Temp7          : Output_List_Node_Ptr;

begin

    Ada.Text_Io.Put_Line(" ");
    Ada.Text_Io.Put_Line("-----------------------------");
    ada.Text_IO.Put_Line("OUTPUTS IN FILE:");

    Temp7 := Output_List;
    loop
       exit when Temp7 = null;
       Put_Line(Wide_String_To_String(Temp7.Output_Name.all,1));
       Temp7 := Temp7.Next;
    end loop;

    Ada.Text_Io.Put_Line("-----------------------------");

end Print_Output_Names;
-------------------------------------------------------------------


-------------------------------------------------------------------
procedure Print_Table_Info(State_Variable_List:State_variable_ptr)is
    Temp  : Row_Ptr                := null;
    Temp1 : Next_State_Node_Ptr    := null;
    Temp2 : Output_Column_Ptr      := null;
    Temp3 : Output_Value_Node_Ptr  := null;
    Temp4 : Conditional_Node_Ptr;
    Temp5 : Conditional_Node_Ptr;
    Temp6       : State_Variable_Ptr;
    Temp7 : Signal_Column_Ptr      := null;
    Temp8 : Signal_Value_Node_Ptr  := null;

begin

    ada.Text_IO.Put_Line(" ");
    ada.Text_IO.Put_Line("---------------------------");
    ada.Text_IO.Put_Line("----- Table Info ---------");
    ada.Text_IO.Put_Line("---------------------------");

    Temp6 := State_Variable_List;
    loop
       exit when (Temp6 = null);
```

```
ada.Text_IO.Put_Line(" ");
Put_Line("State Variable:
"&Wide_String_To_String(Temp6.signal_name.all,1));
Temp := Temp6.truth_table;
loop
   ada.Text_IO.Put_Line(" ");
   exit when Temp = null;
   Put_Line(">State_Name:
   "&Wide_String_To_String(Temp.Present_State.all,1));
   Temp1 := Temp.Next_State_Head_Ptr;
   loop
      exit when Temp1 = null;
      Put_Line("   -Next State Name:
   "&Wide_String_To_String(Temp1.Next_State_Name.all,1));
      Temp4 := Temp1.Conditions;
      loop
         exit when Temp4 = null;
         Put_Line("      "
         &Wide_String_To_String(Temp4.Name.all,1)&":"&
         Wide_String_To_String(Temp4.Value.all,1));
         Temp4 := Temp4.Next;
      end loop;
      Temp1 := Temp1.Next;
   end loop;

   ada.Text_IO.Put_Line(" ");

   Temp2 := Temp.Output_Column_List_Ptr;
   loop
      exit when Temp2 = null;
      Temp3 := Temp2.Output_Value_List_Head_Ptr;
      loop
         exit when Temp3 = null;
         Put("
        "&Wide_String_To_String(Temp2.Output_Name.all,1)&" =
        "&Wide_String_To_String(Temp3.output_value.all,1)&"
         if ( ");
         Temp5 := Temp3.Input_Combinations;
         loop
            exit when Temp5 = null;
            Put(Wide_String_To_String(Temp5.Name.all,1)&":
           "&Wide_String_To_String(Temp5.Value.all,1)&"  ");
            Temp5 := Temp5.Next;
         end loop;
         ada.Text_IO.Put_Line(")");
         Temp3 := Temp3.Next;
      end loop;
      Temp2 := Temp2.Next;
    end loop;

   Temp7 := Temp.Signal_Column_List_Ptr;
   loop
      exit when Temp7 = null;
      Temp8 := Temp7.Signal_Value_List_Head_Ptr;
      loop
         exit when Temp8 = null;
         Put("
```

```
                "&Wide_String_To_String(Temp7.Signal_Name.all,1)&" =
                "&Wide_String_To_String(Temp8.Signal_value.all,1)&"  if
                 ( ");
                  Temp5 := Temp8.Input_Combinations;
                  loop
                     exit when Temp5 = null;
                     Put(Wide_String_To_String(Temp5.Name.all,1)&":
                     "&Wide_String_To_String(Temp5.Value.all,1)&"  ");
                     Temp5 := Temp5.Next;
                  end loop;
                  ada.Text_IO.Put_Line(")");
                  Temp8 := Temp8.Next;
               end loop;
               Temp7 := Temp7.Next;
            end loop;
            Temp := Temp.Next;
         end loop;

         ada.Text_IO.Put_Line("--------------------------");

         Temp6 :=Temp6.next;
      end loop;

   end Print_Table_Info;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   function Itos (
         I : Integer )
     return String is

      Res : String (1 .. 40);
      J   : Integer;

   begin
      Put(Res, I);
      J := Res'Last;    -- Find end of the stupid leading spaces.
      loop
         exit when J < Res'First;
         exit when Res(J) = ' ';
         J := J - 1;
      end loop;
      return Res(J + 1 .. Res'Last);

   end Itos;
   ------------------------------------------------------------------

-----------------------------------------------------------------------
-------------------------------------------------
```

```
PARSING_FUNCTIONS.ADB
-------------------------------------------------------------------

-------------------------------------------------

package body parsing_functions is


   ---------------------------------------------------------------
   procedure Check_For_Comment(Input   :  in
   Ada.Text_Io.File_Type;
                  C   :  in out Character;
                  Flag : out Boolean) is

      Endofline : Boolean   := False;

   begin
         flag := false;
            if (C = '-') then       -- the following section of code is
                                       used to ignore comments
               Get(Input, C);
               Look_Ahead(Input, C, Endofline);
               if (C = '-') then
                  Skip_Line(Input);
                  Look_Ahead(Input, C, Endofline);
               else
                  C := '-';        -- if we do not find a second -, then
                                      we set C to be the first - make
                                      the checking routine invisible
                  Flag := True;
                  return;
               end if;
            else
               return;
            end if;
   end Check_For_Comment;
   ---------------------------------------------------------------


   ---------------------------------------------------------------
      ■  gets the next word, ignores spaces, tabs, parantheses,
         semicolons and colons.
   ---------------------------------------------------------------
   procedure Get_Next_Word (
         Input   :  in    Ada.Text_Io.File_Type;
         Delimiter : in     Character;
         Word : out Lines;
         Length : out Integer;
         Eof       :    out Boolean    )  is

      C         : Character;
      C_Prev    : Character;
      Endofline : Boolean   := False;
      flag : boolean := false;

   begin
      Length := 0;
```

```
   Eof := False;
   loop
      if End_Of_File(Input) then
         Eof := True;
         return;
      elsif End_Of_Line(Input) then
         Skip_Line(Input);
      else
         Look_Ahead(Input, C, Endofline);

         exit when (C /= Delimiter) and (C /= ' ') and (C /=
         Tab) and (C /= '(')and (C /= ')') and (C /= ';') and
        (C /= ':') and (C /= ',')and (C /= '=') and (C /= '<')
        and (C /= '>');
         Get(Input, C);
      end if;
   end loop;

   C_prev := ' ';

   loop
      Look_Ahead(Input, C, Endofline);
      if ((C = '-') and (C_Prev = '-')) then
         Skip_Line(Input);
         Length := Length - 1;
         C_Prev := ' ';
         loop
            if End_Of_File(Input) then
               Eof := True;
               return ;
            elsif End_Of_Line(Input) then
               Skip_Line(Input);
            else
               Look_Ahead(Input, C, Endofline);
               exit when (C /= Delimiter) and (C /= ' ') and
               (C /= Tab) and (C /= '(')and (C /= ')') and (C
                /= ';') and (C /= ':') and (C /= ',')and (C /=
               '=') and (C /= '<') and (C /= '>');
               Get(Input, C);
            end if;
         end loop;
      else
         C_Prev := C;
         exit when (C = Delimiter) or (C = ' ') or (C = Tab)
         or (C = '(')or (C = ')')  or (C = ';') or (C = ':')
         or (C = ',')or (C = '=') or (C = '<') or (C = '>') or
        (C = '/')  ; --JUST ADDED 26th APRIL
         Get(Input, C);
         Length := Length + 1;
         -- Put(c);
         Word(Length) := C;
         exit when End_Of_Line(Input);
      end if;
   end loop;

end Get_Next_Word;
-------------------------------------------------------------
```

```
-------------------------------------------------------------
function Next_Word (
      Input  :  in    Ada.Text_Io.File_Type;
      Delimiter : in     Character) return Wide_String_Ptr is

   C         : Character;
   C_prev : character;
   C1         : Character;
   Endofline : Boolean   := False;
   Eof : Boolean;
   Length : integer := 0;
   Word             :          Lines;
   flag : boolean := false;

begin
   Eof := False;
   loop
      if End_Of_File(Input) then
         Eof := True;
         return NULL;
      elsif End_Of_Line(Input) then
         Skip_Line(Input);
      else
         Look_Ahead(Input, C, Endofline);
         exit when (C /= Delimiter) and (C /= ' ') and (C /=
         Tab) and (C /= '(')and (C /= ')') and (C /= ';') and
         (C /= ':') and (C /= ',')and (C /= '=') and (C /= '<')
         and (C /= '>');
         Get(Input, C);
      end if;
   end loop;


   C_prev := ' ';

   loop
      Get(Input, C);
      if (C = '-' and C_Prev = '-') then
         Skip_Line(Input);
         Length := Length - 1;
         C_Prev := ' ';
         loop
            if End_Of_File(Input) then
               Eof := True;
               return NULL;
            elsif End_Of_Line(Input) then
               Skip_Line(Input);
            else
               Look_Ahead(Input, C, Endofline);
               exit when (C /= Delimiter) and (C /= ' ') and
               (C /= Tab) and (C /= '(')and (C /= ')') and (C
                /= ';') and (C /= ':') and (C /= ',')
                and (C /= '=') and (C /= '<') and (C /= '>')  ;
               Get(Input, C);
            end if;
```

```
            end loop;
         else
            C_Prev := C;
            exit when (C = Delimiter) or (C = ' ') or (C = Tab)
            or (C = '(')or (C = ')')  or (C = ';') or (C = ':')
            or (C = ',')or (C = '=') or (C = '<') or (C = '>') or
           (C = '/'); -- ADDED APRIL 26th

         Length := Length + 1;
         -- Put(c);
         Word(Length) := C;
         exit when End_Of_Line(Input);
         end if;
      end loop;


      return new
Wide_String'(characters.handling.To_Wide_String(Word(1..Length)));

   end Next_Word;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   function Get_Next_Operator (
   Input  :  in    Ada.Text_Io.File_Type) return
   Wide_String_Ptr is

      C         : Character;
      C_prev : character;
      C1          : Character;
      Endofline : Boolean   := False;
      Eof : Boolean;
      Length : integer := 0;
      Word              :          Lines;
      flag : boolean := false;

   begin
      Eof := False;
      loop
         if End_Of_File(Input) then
            Eof := True;
            return NULL;
         elsif End_Of_Line(Input) then
            Skip_Line(Input);
         else
            Look_Ahead(Input, C, Endofline);
            exit when (C /= ' ') and (C /= Tab) and (C /= '(')
            and (C /= ')') and (C /= ';') and (C /= ':') and (C
            /= ',');
            Get(Input, C);
         end if;
      end loop;


      C_prev := ' ';
```

```
      loop
         Look_Ahead(Input, C, Endofline);
         if (C = '-' and C_Prev = '-') then
            Skip_Line(Input);
            Length := Length - 1;
            C_Prev := ' ';
            loop
               if End_Of_File(Input) then
                  Eof := True;
                  return NULL;
               elsif End_Of_Line(Input) then
                  Skip_Line(Input);
               else
                  Look_Ahead(Input, C, Endofline);
                  exit when (C /= ' ') and (C /= Tab) and (C /=
                  '(') and (C /= ')') and (C /= ';') and (C /=
                   ':') and (C /= ',');
                  Get(Input, C);
               end if;
            end loop;
         else
            C_Prev := C;
            exit when (C /= '=') and (C /= '<') and (C /= '>')
             and (C /= '/');
            Get(Input, C);
            Length := Length + 1;
            -- Put(c);
            Word(Length) := C;
            exit when End_Of_Line(Input);
         end if;
      end loop;


      return new
Wide_String'(characters.handling.To_Wide_String(Word(1..Length)));

   end Get_Next_Operator;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
      ■ gets the next word using the passed in variable as a delimiter
        as well as spaces, commas and tabs
      ■ when it encounters the delimiter itself, it returns the
        delimiter as the next word
   ------------------------------------------------------------------
   procedure Special_Get_Next_Word (
         Input   :  in    Ada.Text_Io.File_Type;
         Delimiter: in Character;
         Word       :    out Lines;
         Length     :    out Integer;
         Eof        :    out Boolean    ) is

      C          : Character;
      C1          : Character;
      Endofline : Boolean   := False;
      flag : boolean := false;
```

```
begin
   Length := 0;
   Eof := False;
   loop
      if End_Of_File(Input) then
         Eof := True;
         return;
      elsif End_Of_Line(Input) then
         Skip_Line(Input);
      else
         Look_Ahead(Input, C, Endofline);

         Check_For_Comment(Input,C,flag);

         exit when (C /= ',') and (C /= ' ') and (C /= Tab)
         and (C /= '(');
         Get(Input, C);
      end if;
   end loop;

   Look_Ahead(Input, C, Endofline);
   if (C = Delimiter) then
      Get(Input,C);
      Length := 1;
      Word(Length) := C;
      return;
   end if;

   loop
      Look_Ahead(Input, C, Endofline);
      exit when (C = ',') or (C = ' ') or (C = Tab) or (C = '(')
      or (C = Delimiter);
      Get(Input,C);
      Length := Length + 1;
      Word(Length) := C;
      exit when End_Of_Line(Input);
   end loop;

end Special_Get_Next_Word;
-------------------------------------------------------------------


-------------------------------------------------------------------
procedure Get_until (
      Input   :  in    Ada.Text_Io.File_Type;
      Delimiter: in Character;
      Word       :    out Lines;
      Length     :    out Integer;
      Eof        :    out Boolean    ) is

   C          : Character;
   C1          : Character;
   Endofline : Boolean   := False;
   flag : boolean := false;

begin
```

```
   Length := 0;
   Eof := False;
   loop
      if End_Of_File(Input) then
         Eof := True;
         return;
      elsif End_Of_Line(Input) then
         Skip_Line(Input);
      else
         Look_Ahead(Input, C, Endofline);

         Check_For_Comment(Input,C,flag);

         exit when (C /= Delimiter) and (C /= ''') and (C /= '
         ') and (C /= Tab) and (C /= '(')and (C /= ')')  and
         (C /= ':') and (C /= ',')and (C /= '=') and (C /= '<')
         and (C /= '>') and (C /= '&');
         Get(Input, C);
      end if;
   end loop;

   if (Flag) then
       C := '-';
    else
       Get(Input, C);
   end if;

   loop
      exit when (C = Delimiter);
      if((C /= ''') and (C /= Tab) and (C /= '(')and (C /= ')')
      and (C /= '=') and (C /= '<') and (C /= '>') and (C /= '
      ') and (C /= '&')) then
         Length := Length + 1;
         -- Put(c);
         if(Length > Maximum_String_Length) then
            Length := 1;
         end if;
         Word(Length) := C;
         exit when End_Of_Line(Input);
       end if;
      Get(Input, C);
   end loop;

end Get_Until;
-------------------------------------------------------------------


-------------------------------------------------------------------
procedure Find_Next_matching_character (
      Input  :  in    Ada.Text_Io.File_Type;
      Char: in Character  ) is

   C         : Character;
   C1         : Character;
   Endofline : Boolean   := False;
   flag : boolean := false;
   Eof : Boolean;
```

```
begin
   Eof := False;
   loop
      if End_Of_File(Input) then
         Eof := True;
         return;
      elsif End_Of_Line(Input) then
         Skip_Line(Input);
      else
         Look_Ahead(Input, C, Endofline);

         Check_For_Comment(Input,C,flag);
         Put(c);
         exit when (C = Char);
         Get(Input, C);
      end if;
   end loop;

   Get(Input, C);

end Find_next_matching_Character;
-------------------------------------------------------------------


-------------------------------------------------------------------
   ■  Finds first occurrence of a word in a file
-------------------------------------------------------------------
procedure Find_Word (
      Input : in out  Ada.Text_Io.File_Type;
      Search_Word : in String;
      eof : out boolean )  is

   Word        : Lines;
   Word_Length : Integer;

begin
   Reset(Input, In_File);
   eof := false;
   loop
      Get_Next_Word(Input,' ',Word, Word_Length, Eof);
      exit when (To_Upper(Word(1..Word_Length)) =
   To_Upper(Search_Word)) or eof;
   end loop;
end Find_Word;
-------------------------------------------------------------------


-------------------------------------------------------------------
   ■  Finds the next matching word from the last word read
-------------------------------------------------------------------
function Find_Next_Matching_Word (
      Input  :  in    Ada.Text_Io.File_Type;
      Search_Word : in     String ) return integer is

   Word        : Lines;
   Word_Length : Integer;
```

```
   Eof         : Boolean;

begin
   loop
      Get_Next_Word(Input,' ',Word, Word_Length, Eof);
      exit when (To_Upper(Word(1..Word_Length)) =
      To_Upper(Search_Word)) or eof;
   end loop;

   if(Eof) then
      return -1;
   else
      return 0;
   end if;

end Find_Next_Matching_Word;
-------------------------------------------------------------


-------------------------------------------------------------
procedure Get_Side_Bracket(Input  :  in
Ada.Text_Io.File_Type) is
   C : Character;
begin
   loop
      Get(Input, C);
      exit when C = '(';
   end loop;
end Get_Side_Bracket;
-------------------------------------------------------------


-------------------------------------------------------------
procedure Find_Next_Conditional_Adjective (
      Input  :  in    Ada.Text_Io.File_Type;
      Word        :    out Lines;
      Word_Length :    out Integer;
      Eof         :    out Boolean  ) is

begin
   loop
      Get_Next_Word(Input,' ',Word, Word_Length, Eof);
      exit when eof;
      if (Compare_Words(Word(1..Word_Length),"IF") or
   Compare_Words(Word(1..Word_Length),"ELSE") or
   Compare_Words(Word(1..Word_Length),"ELSIF") or
   Compare_Words(Word(1..Word_Length),"WHEN") or
   Compare_Words(Word(1..Word_Length),"END")) then
         return;
      end if;
   end loop;

end Find_Next_Conditional_Adjective;
-------------------------------------------------------------
```

```
-------------------------------------------------------------------------
procedure Find_Next_Variable_Or_Then (
      Input  :  in    Ada.Text_Io.File_Type;
      Input_List_Head_Ptr : in Input_List_Node_Ptr;
      Signal_List : in Signal_List_Node_Ptr;
      Word        :    out Lines;
      Word_Length :    out Integer;
      Eof         :    out Boolean;
      type_of_conditional_variable : out Conditional_Node_Type
       ) is

   Flag : Boolean             := False;
   Ptr1  : Input_List_Node_Ptr;
   Ptr2 : Signal_List_Node_Ptr;
   temp_word : Wide_String_Ptr;

begin
   loop
      Get_Next_Word (Input,'=',Word,Word_Length,Eof);
      if(Eof) then
            Ada.Text_Io.Put_Line("ERROR:
            Find_next_variable_or_then : reached eof and did
            not find THEN or variable");
      end if;
      exit when Compare_Words(Word(1..Word_Length),"THEN") or
       eof;

      Temp_Word := new Wide_String'
     (Characters.Handling.To_Wide_String(Word(1..Word_Length)));

      Ptr1:= Input_List_Head_Ptr;
      loop
         exit when (Ptr1 = null);
         if
         (Compare_Words(Wide_String_To_String(Ptr1.Input_Name.All,1),
         Wide_String_To_String(Temp_Word.All,1))) then
            type_of_Conditional_variable :=  Input_Conditional;
            return;
         end if;
         Ptr1 := Ptr1.Next;
      end loop;

      Ptr2 := Signal_List;
      loop
         exit when (Ptr2 = null);
         if
        (Compare_Words(Wide_String_To_String(Ptr2.Signal_Name.All,1),
        Wide_String_To_String(Temp_Word.All,1))) then
            type_of_Conditional_variable :=  Signal_Conditional;
            return;
         end if;
         Ptr2 := Ptr2.Next;
      end loop;

   end loop;

end Find_Next_Variable_OR_then;
```

```
   ------------------------------------------------------------------------


   ------------------------------------------------------------------------
   procedure Find_Next_State_Name (
         Input   :  in    Ada.Text_Io.File_Type;
         Word          :    out Lines;
         Word_Length :    out Integer;
         Eof           :    out Boolean;
         Row_Head : Row_Ptr  ) is

      Flag : Boolean := False;
      Ptr  : Row_Ptr := Row_Head;
      temp_word : Wide_String_Ptr;

   begin
      loop
         Get_Next_Word(Input,' ',Word, Word_Length, Eof);
         Ptr:= Row_Head;
         loop
            exit when Ptr = null;
            temp_word := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
            if (Ptr.Present_State.all = temp_word.all)then
               return;
            end if;
            Ptr := Ptr.Next;
         end loop;
      end loop;

   end Find_Next_State_Name;
   ------------------------------------------------------------------------


   ------------------------------------------------------------------------
   function Compare_Words(
         String1 : String;
         String2 : String) return Boolean is
   begin
      if (To_Upper(String1) = To_Upper(String2)) then
         return True;
      else
         return False;
      end if;
   end Compare_Words;
   ------------------------------------------------------------------------


   ------------------------------------------------------------------------
   procedure Find_End_For(
      Input   :  in    Ada.Text_Io.File_Type;
      String1 : String) is

      first_Word : wide_String_ptr;

   begin
      loop
         First_Word := Next_Word(Input,' ');
```

```
            if
(Compare_Words(Wide_String_To_String(First_Word.All,1),String1)) then
               Find_End_For(Input,String1);
            elsif
(Compare_Words(Wide_String_To_String(First_Word.All,1),"END")) then
               First_Word := Next_Word(Input,' ');
               if
(Compare_Words(Wide_String_To_String(First_Word.All,1),String1)) then
                  exit;
               end if;
            end if;
         end loop;
   end Find_End_for;
   ------------------------------------------------------------------


end Parsing_Functions;
-----------------------------------------------------------------------
-------------------------------------------------
```

STATEMENT_HANDLERS.ADB

```
-----------------------------------------------------------------------
-------------------------------------------------
package body Statement_Handlers is

   ------------------------------------------------------------------
   procedure Process_If_Statement (
          Input  :  in    Ada.Text_Io.File_Type;
          State_Info: Signal_Name_Value_Node_Ptr;
          Conditional_List: Conditional_Node_Ptr;
          Input_List_Head_Ptr: Input_List_Node_Ptr;
          Signal_List : Signal_List_Node_Ptr;
          Output_List_Head_Ptr: Output_List_Node_Ptr) is

      Flag : Integer;
      New_Conditional_List : Conditional_Node_Ptr;
      temp_Conditional_List : Conditional_Node_Ptr;
      Temp_Ptr : Conditional_Node_Ptr;
      Name                 : Wide_String_Ptr;
      Operator                 : Wide_String_Ptr;
      Value                : Wide_String_Ptr;
      Last                      : Positive;
      Next_State                : Wide_String_Ptr;
      Type_Of_Conditional_Variable :   Conditional_Node_Type;
      temp_int : integer;
      Word             :          Lines;
      Word_Length      :          Integer;
      Eof              :          Boolean;


      negation_list : conditional_node_ptr;
      negative_Value               : Wide_String_Ptr;
      Negation : Wide_String_Ptr := new
      Wide_String'(Characters.Handling.To_Wide_String("*"));
```

```
      Counter : Integer;
      --New_State_Info : Signal_Name_Value_Node_Ptr;

    begin
      --New_State_Info := new Signal_Name_Value_Node;
      ---New_State_Info.Signal_Name := State_Info.Signal_Name;
      --New_State_Info.Signal_Value := State_Info.Signal_Value;
      --New_State_Info.priority := 0;
      ada.Text_IO.Put_Line("BEGNING IF HANDLER");

      Copy_Conditional_List(Conditional_List,New_Conditional_List);

      loop

Find_Next_Variable_Or_Then(Input,Input_List_Head_Ptr,Signal_List,Word,
Word_Length,Eof,type_of_conditional_variable);
         exit when Compare_Words(Word(1..Word_Length),"THEN");
         Name := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
         Ada.Text_Io.Put_Line("If Conditional: "&Word(1..Word_Length));
         Operator := Get_Next_Operator (Input);    -- we are getting the
OPERATOR sign
         Ada.Text_Io.Put_Line("Operator:
'"&Wide_String_To_String(operator.all,1)&"'");
         Get_Next_Word(Input,''',Word, Word_Length, Eof);
         Value := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
         Ada.Text_Io.Put_Line("Value:
'"&Wide_String_To_String(value.all,1)&"'");
         Append_Element_To_Conditional_List
(New_Conditional_List,Type_Of_Conditional_Variable,Name,Operator,Value);
         negative_value := new
Wide_String'(Characters.Handling.To_Wide_String("*"&Wide_String_To_Stri
ng(name.All,1)));
         Append_Element_To_Conditional_List
(negation_list,Type_Of_Conditional_Variable,negative_value,Operator,Val
ue);
      end loop;

      State_Info.priority := State_Info.priority + 1;

      loop
         Temp_Int :=
         Process_Statement(Input,State_Info,New_Conditional_List,
         Input_List_Head_Ptr,Signal_List,Output_List_Head_Ptr);
         if (Temp_Int = 3) then
            exit;
         elsif (Temp_Int = 2) then
            exit;
         elsif (Temp_Int = -1) then
            Get_Next_Word(Input,' ',Word, Word_Length, Eof); -- getting
            the if part in "END IF"
            ada.Text_IO.Put_Line(":::"&word(1..word_length));
            return;
         end if;
       end loop;
```

```
      loop
         Copy_Conditional_List(Conditional_List,New_Conditional_List);
         Copy_Conditional_List(negation_List,Temp_conditional_list);
         if (Temp_int = 2) then      -- elsif
            loop

Find_Next_Variable_Or_Then(Input,Input_List_Head_Ptr,Signal_List,Word,W
ord_Length,Eof,type_of_conditional_variable);
               exit when Compare_Words(Word(1..Word_Length),"THEN");
               Name := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
               Ada.Text_Io.Put_Line("Elsif Conditional:
"&Word(1..Word_Length));
               Operator := Get_Next_Operator (Input);    -- we are
getting the OPERATOR sign
               Ada.Text_Io.Put_Line("Operator:
'"&Wide_String_To_String(operator.all,1)&"'");
               Get_Next_Word(Input,''',Word, Word_Length, Eof);
               value := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
               Append_Element_To_Conditional_List
(New_Conditional_List,Type_Of_Conditional_Variable,Name,Operator,Value);
               negative_value := new
Wide_String'(Characters.Handling.To_Wide_String("*"&Wide_String_To_Stri
ng(name.All,1)));
               Append_Element_To_Conditional_List
(negation_list,Type_Of_Conditional_Variable,negative_value,Operator,Val
ue);
            end loop;

Add_Conditional_Lists(New_Conditional_List,Temp_Conditional_List);
            State_Info.priority := State_Info.priority + 1;
            loop
               Temp_Int :=
Process_Statement(Input,State_Info,New_Conditional_List,Input_List_Head
_Ptr,Signal_List,Output_List_Head_Ptr);
               if (Temp_Int = 3) then
                  exit;
               elsif (Temp_Int = 2) then
                  exit;
               elsif (Temp_Int = -1) then
                  Get_Next_Word(Input,' ',Word, Word_Length, Eof); --
getting the if part in "END IF"
                  return;
               end if;
             end loop;

          elsif (Temp_Int = 3) then       -- else

Copy_Conditional_List(Conditional_List,New_Conditional_List);
            Copy_Conditional_List(negation_List,Temp_conditional_list);

Add_Conditional_Lists(New_Conditional_List,Temp_Conditional_List);
             State_Info.priority := State_Info.priority + 1;
            loop
```

```
               Temp_Int :=
Process_Statement(Input,State_Info,new_Conditional_List,Input_List_Head
_Ptr,Signal_List,Output_List_Head_Ptr);
               if (Temp_Int = 3) then
                  exit;
               elsif (Temp_Int = 2) then
                  exit;
               elsif (Temp_Int = -1) then
                  Get_Next_Word(Input,' ',Word, Word_Length, Eof); --
                  getting the if part in "END IF"
                  return;
               end if;
             end loop;

         else
            Ada.Text_Io.Put_Line("ERROR --- NO END FOR IF STATEMENT
            FOUND !!!!");
            return;
         end if;
      end loop;


   end Process_IF_Statement;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
    procedure Process_State_Assignment_Statement(
      Input  :  in    Ada.Text_Io.File_Type;
      State_Info: Signal_Name_Value_Node_Ptr;
      Conditional_List: in Conditional_Node_Ptr;
      First_Word: Wide_String_Ptr ) is

      Value : Wide_String_Ptr;
      temp_int : integer;
      Word              :          Lines;
      Word_Length       :          Integer;
      Eof               :          Boolean;

   begin

      --Get_Next_Word(Input,' ',Word, Word_Length, Eof);     -- we are
getting the "<=" sign
      Get_until(Input,';',word,word_length,eof);
      Value := new
Wide_String'(characters.handling.To_Wide_String(Word(1..Word_Length)));
-- we are getting assignment Value
      Ada.Text_Io.Put_Line("IN PROCESS STATE ASSIGNMENT,
"&Wide_String_To_String(State_Info.Signal_name.All,1)&" =
"&Wide_String_To_String(Value.All,1));
      if(just_negation(Conditional_List) and
Compare_words(Wide_String_To_String(Value.All,1),Wide_String_To_String(
State_Info.Signal_Value.All,1))) then
         Ada.Text_Io.Put_Line("FOUND UNCONDITIONAL SELF LOOP ::::
"&Wide_String_To_String(Value.All,1));
      else
```

```
Fill_In_Transition(State_Info,Value,Conditional_List,State_Variable_lis
t);
      end if;
      --       Find_Next_Matching_Character(Input,';');

   end Process_State_Assignment_Statement;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   procedure Process_Output_Assignment_Statement(
      Input  :  in    Ada.Text_Io.File_Type;
      State_Info: Signal_Name_Value_Node_Ptr;
      Conditional_List: in Conditional_Node_Ptr;
      First_Word: Wide_String_Ptr) is

      Output_Value : Wide_String_Ptr;
      temp_int : integer;
      Word              :          Lines;
      Word_Length       :          Integer;
      Eof               :          Boolean;

   begin

      --Get_Next_Word(Input,' ',Word, Word_Length, Eof);     -- we are
getting the "<=" sign
      Get_until(Input,';',word,word_length,eof);
      Output_Value := new
Wide_String'(characters.handling.To_Wide_String(Word(1..Word_Length)));
-- we are getting assignment Value

Fill_In_Output_Value(State_Info,First_Word,Output_Value,Conditional_Lis
t,State_Variable_List);
      Ada.Text_Io.Put_Line("OUTPUT VALUE FROM ASSIGMENT STATEMENT:
"&Wide_String_To_String(Output_Value.All,1));
--       Find_Next_Matching_Character(Input,';');

   end Process_Output_Assignment_Statement;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   procedure Process_Signal_Assignment_Statement(
      Input  :  in    Ada.Text_Io.File_Type;
      State_Info: Signal_Name_Value_Node_Ptr;
      Conditional_List: in Conditional_Node_Ptr;
      First_Word: Wide_String_Ptr) is

      Signal_Value : Wide_String_Ptr;
      temp_int : integer;
      Word              :          Lines;
      Word_Length       :          Integer;
      Eof               :          Boolean;

   begin
```

```
      --Get_Next_Word(Input,' ',Word, Word_Length, Eof);     -- we are
        getting the "<=" sign
      Get_until(Input,';',word,word_length,eof);
      Signal_Value := new
Wide_String'(characters.handling.To_Wide_String(Word(1..Word_Length)));
-- we are getting assignment Value
      Fill_In_Signal_Value(State_Info,First_Word,Signal_Value,
      Conditional_List,State_Variable_List);
      Ada.Text_Io.Put_Line("SIGNAL VALUE FROM ASSIGMENT STATEMENT:
      "&Wide_String_To_String(signal_Value.All,1));
       --       Find_Next_Matching_Character(Input,';');

   end Process_Signal_Assignment_Statement;
   ------------------------------------------------------------


   ------------------------------------------------------------
   function Process_Statement(
      Input  :  in    Ada.Text_Io.File_Type;
      State_Info: Signal_Name_Value_Node_Ptr;
      Conditional_List:  Conditional_Node_Ptr;
      Input_List_Head_Ptr: Input_List_Node_Ptr;
      Signal_List : Signal_List_Node_Ptr;
      Output_List_Head_Ptr: Output_List_Node_Ptr ) return integer is

      S_Type : Statement_Type;
      First_Word : Wide_String_Ptr;
      New_Conditional_List : Conditional_Node_Ptr;
      State_Variable_Name: Wide_String_Ptr;
      Word              :          Lines;
      Word_Length       :          Integer;
      Eof               :          Boolean;

   begin


      Copy_Conditional_List(Conditional_list,New_Conditional_List);

      First_Word := Next_Word(Input,' ');
      ada.Text_IO.Put_Line("PROCESS STATEMENT: FIRST WORD =
"&Wide_String_To_String(First_Word.all,1));
      if(First_Word = null) then
         return -1;
      end if;


      if  (Compare_Words(Wide_String_To_String(First_Word.all,1),"WHEN"))
      then
         return 0;

elsif(Compare_Words(Wide_String_To_String(First_Word.all,1),"END"))
then
         return -1;

elsif(Compare_Words(Wide_String_To_String(First_Word.all,1),"ELSIF"))
then
         return 2;
```

```
elsif(Compare_Words(Wide_String_To_String(First_Word.all,1),"ELSE"))
then
         return 3;
      end if;

      S_Type :=
Type_Of_Statement(State_Info,First_Word,Signal_List,Input_List_Head_Ptr
,Output_List_Head_Ptr);

      case S_Type is
         when If_Statement =>

Process_If_Statement(Input,State_Info,New_Conditional_List,Input_List_H
ead_Ptr,Signal_List,Output_List_Head_Ptr);
         when Case_Statement =>

Process_Case_Statement(Input,State_Info,New_Conditional_List,State_vari
able_name,Signal_List,Input_List_Head_Ptr,Output_List_Head_Ptr);
         when State_Assignment_Statement =>

Process_State_Assignment_Statement(input,State_Info,New_Conditional_Lis
t,First_Word);
         when Signal_Assignment_Statement =>

Process_Signal_Assignment_Statement(Input,State_Info,New_Conditional_li
st,First_Word);
         when Output_Assignment_Statement =>

Process_Output_Assignment_Statement(Input,State_Info,New_Conditional_li
st,First_Word);
         when others =>
              Ada.Text_Io.Put_Line("Unknown Statement Encountered");
              Get_Until(Input,';',Word,Word_Length,Eof);   -- IGNOR
REST OF LINE!!!!!
      end case;

      return 1;

   end Process_Statement;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   procedure Process_Case_Statement(
      Input  :  in    Ada.Text_Io.File_Type;
      State_Info: Signal_Name_Value_Node_Ptr;
      Conditional_List: Conditional_Node_Ptr;
      State_Variable_Name: out Wide_String_Ptr;
      Signal_List : Signal_List_Node_Ptr;
      Input_List_Head_Ptr: Input_List_Node_Ptr;
      Output_List_Head_Ptr: Output_List_Node_Ptr )  is

      Conditional   : Wide_String_Ptr;
      Variable_Type : Conditional_Type;
      Flag : Integer;
      State_Value : Wide_String_Ptr;
```

```
      New_State_Info : Signal_Name_Value_Node_Ptr;
      New_List                : Conditional_Node_Ptr;
      New_Conditional_List : Conditional_Node_Ptr;
      Temp_Ptr : Conditional_Node_Ptr;
      Temp_Int : Integer;
      Eof : Boolean;
      Word_Length : Integer;
      WOrd : lines;

   begin
        conditional :=  Next_Word(Input,' ');
    --    Get_Next_Word(Input,' ',Word, Word_Length, Eof);
      --    Conditional := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
      Ada.Text_Io.Put_Line(" ");
      ada.Text_IO.Put_Line("----------------------------------------
---------");
      ada.Text_IO.Put_Line("STARTING CASE STATEMENT WITH:
"&Wide_String_To_String(conditional.all,1));
      Variable_Type :=
Type_Of_Conditional(Conditional,Signal_List,Input_List_Head_Ptr,Output_
List_Head_Ptr);
      case Variable_Type is
         when State_Variable =>
            Get_Next_Word(Input,' ',Word, Word_Length, Eof); -- get IS
            Get_Next_Word(Input,' ',Word, Word_Length, Eof); -- get
First WHEN
            loop
                  Get_Next_Word(Input,' ',Word, Word_Length, Eof);
                  State_Value := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
                  ada.Text_IO.put_line("WHEN CONDIITONAL :
"&Wide_String_To_String(state_value.all,1));
               --  Get_Next_Word(Input,' ',Word, Word_Length, Eof);  -
- getting '=>' sign

                  if (State_Info = null) then
                     Create_State_Variable_Entry(
                     State_Variable_List,Conditional);
                     Add_Row(Output_List_Head_Ptr,Signal_List,
                     Conditional,State_Value,State_Variable_List);
                     New_State_Info := new Signal_Name_Value_Node;
                     New_State_Info.Signal_Name := Conditional;
                     New_State_Info.Signal_Value := State_Value;
                     New_State_Info.priority := 0;
                     state_variable_name := conditional;
                     loop
                        Temp_Int :=
Process_Statement(Input,New_State_Info,Conditional_List,Input_List_Head
_Ptr,Signal_List,Output_List_Head_Ptr);
                        if (Temp_Int = 0) then
                              exit;
                        elsif (Temp_Int = -1) then
                              Get_Next_Word(Input,' ',Word,  Word_Length,
Eof); -- found END, now getting CASE in "END CASE"
                              Ada.Text_Io.Put_Line("FOUND END FOR:
"&Wide_String_To_String(New_State_Info.Signal_Name.All,1));
```

```
                                ada.Text_IO.Put_Line("-------------------
---------------------------------");
                                return;
                             end if;
                          end loop;

                       else
                          Ada.Text_Io.Put_Line("This WHEN is for a NESTED
CASE!!");
                          New_Conditional_List :=
Add_Element_To_Conditional_List
(Conditional_List,Signal_Conditional,Conditional,new
Wide_String'(Characters.Handling.To_Wide_String("=")),State_Value);  --
add new state info to condional list
                          loop
                             Temp_Int :=
Process_Statement(Input,State_Info,New_Conditional_List,Input_List_Head
_Ptr,Signal_List,Output_List_Head_Ptr);
                             if (Temp_Int = 0) then
                                   exit;
                             elsif (Temp_Int = -1) then
                                Get_Next_Word(Input,' ',Word, Word_Length,
Eof); -- found END, now getting CASE in "END CASE"
                                Ada.Text_Io.Put_Line("FOUND *END FOR:
"&Wide_String_To_String(conditional.all,1));
                                ada.Text_IO.Put_Line("-------------------
---------------------------------");
                                return;
                             end if;
                          end loop;
                       end if;
                    end loop;
               when others =>
                  Ada.Text_Io.Put_Line("VARIABLE TYPE FOUND NOT YET
                  SUPPORTED");
                  Find_End_For(Input,"Case");
            end case;



      end Process_Case_Statement;
      ------------------------------------------------------------------


      ------------------------------------------------------------------
      function Get_State_Equivalent(State : Wide_String_Ptr) return
      Wide_String_Ptr is

         Temp : Signal_Equivalency_List_Ptr;

      begin
         Temp:= State_Equivalency_List;
         loop
            exit when Temp = null;
            if  (Compare_Words(Wide_String_To_String(Temp.Signal_Name.all,1)
                ,Wide_String_To_String(State.all,1))) then
               return Temp.Equivalent_Signal;
```

```
         end if;
         Temp := Temp.next;
      end loop;
      return null;
   end Get_State_Equivalent;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   function Type_Of_Statement(
      State_Info: Signal_Name_Value_Node_Ptr;
      First_Word : in     Wide_String_Ptr;
      Signal_List : Signal_List_Node_Ptr;
      Input_List_Head_Ptr: Input_List_Node_Ptr;
      Output_List_Head_Ptr: Output_List_Node_Ptr)  return
      Statement_Type is

      Variable_Type : Conditional_Type;
      temp : wide_String_ptr;
   begin

      If (Compare_Words(Wide_String_To_String(First_Word.all,1),"IF"))
      then
         return if_statement;
      elsif
(Compare_Words(Wide_String_To_String(First_Word.all,1),"CASE")) then
         return Case_Statement;
      else
         Variable_Type := Type_Of_Conditional
(First_Word,Signal_List,Input_List_Head_Ptr,Output_List_Head_Ptr);
         case Variable_Type is
            when State_Variable =>
               Temp := Get_State_Equivalent(State_Info.Signal_Name);
               ada.Text_IO.Put_Line("Finding EQUIVALENT STATE FOR
"&Wide_String_To_String(State_Info.signal_name.all,1));
               if (Temp /= null) then
                  ada.Text_IO.Put_Line("FOUND
"&Wide_String_To_String(Temp.all,1));
               end if;

if(Compare_Words(Wide_String_To_String(State_Info.signal_name.all,1),Wi
de_String_To_String(First_Word.all,1))) then
                  return State_Assignment_Statement;
               elsif(Temp /= null) then
                  ada.Text_IO.Put_Line("FOUND EQUIVALENT STATE FOR
"&Wide_String_To_String(State_Info.signal_name.all,1)&" =
"&Wide_String_To_String(Temp.all,1));

if(Compare_Words(Wide_String_To_String(temp.all,1),Wide_String_To_Strin
g(First_Word.all,1))) then
                     return State_Assignment_Statement;
                  else
                     return Signal_Assignment_Statement;
                  end if;
               else
                  return Signal_Assignment_Statement;
               end if;
```

```
            when Output_Variable =>
               return Output_Assignment_Statement;
            when others =>
               return Unknown_Statement;
         end case;
      end if;


   end Type_Of_Statement;
   ------------------------------------------------------------------


end Statement_Handlers;
-------------------------------------------------------------------------
-------------------------------------------------


TABLE_SETUP.ADB
-------------------------------------------------------------------------

-------------------------------------------------

package body Table_Setup_Functions is


   ------------------------------------------------------------------
      ■ This creates the Signal list which contains the names of all
        the Signal variables
   ------------------------------------------------------------------
   procedure Input_Output_Variables (Input  : in out
   Ada.Text_Io.File_Type; Input_List_Head_Ptr : in out
   Input_List_Node_Ptr; Output_List_Head_Ptr : out Output_List_Node_Ptr)
   is
      Word        : Lines;
      Word_Length : Integer;
      Eof         : Boolean;
      temporary_head1 : Input_List_Node_Ptr;
      temporary_head2 : Output_List_Node_Ptr;

      Temp2 : Output_List_Node_Ptr := null;
      tail2  : Output_List_Node_Ptr := null;

      Temp1    : Input_List_Node_Ptr := null;
      tail1     : Input_List_Node_Ptr := null;
      Counter : Integer             := 0;    --- used to caluclate
                                                input_list_length
      Temp_Int: integer := 0;

   begin
      -- now we get the signal variable names
      Reset(Input, In_File);
      loop
         Temp_Int := Find_Next_Matching_Word(Input,"PORT");
```

```
         exit when Temp_Int = -1;

         loop
            Special_Get_Next_Word(Input,':',Word,Word_Length, Eof);
            ada.Text_IO.Put_Line("NEW INPUT OR OUTPUT NAME FOUND:
"&word(1..word_length));

            exit when Compare_Words(Word(1..Word_Length),"map");
            exit when Compare_Words(Word(1..Word_Length),"end");
            Temp1 := new Input_List_Node;
            Temp2 := new Output_List_Node;
            Temp1.Input_Name := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
            Temp2.Output_Name := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
            Temp1.Next := null;
            Temp2.Next := null;

            Temporary_Head1 := Temp1;
            temporary_head2 := temp2;

            Tail1 := Temp1;
            Tail2 := Temp2;

            loop
               Special_Get_Next_Word(Input,':',Word, Word_Length, Eof);
               exit when Word(1..Word_Length) = ":";
               Temp1 := new Input_List_Node;
               Temp2 := new Output_List_Node;
               Temp1.Input_Name := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
               Temp2.Output_Name := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
               Temp1.Next := null;
               Temp2.Next := null;

               tail1.Next := Temp1;
               Tail1 := Temp1;
               tail2.Next := Temp2;
               Tail2 := Temp2;

            end loop;

            Get_Next_Word(Input,' ',Word, Word_Length, Eof);
            if (Compare_Words(Word(1..Word_Length),"IN")) then
               Tail1.Next := Input_List_Head_Ptr;
               Input_List_Head_Ptr := Temporary_Head1;
            elsif
               (Compare_Words(Word(1..Word_Length),"OUT")) then
               Tail2.Next := Output_List_Head_Ptr;
               Output_List_Head_Ptr := Temporary_Head2;
            end if;

            loop
               Special_Get_Next_Word(Input,';',Word,Word_Length, Eof);
               exit when Word(1..Word_Length) = ";";
            end loop;
```

```
      end loop;

   end loop;

end Input_Output_Variables;
-------------------------------------------------------------------



-------------------------------------------------------------------
   ■ This creates the Signal list which contains the names of all
     the Signal variables
-------------------------------------------------------------------
procedure Signals (Input  : in out   Ada.Text_Io.File_Type;
Signal_List : out Signal_List_Node_Ptr ) is
   Word         : Lines;
   Word_Length : Integer;
   Eof          : Boolean;
   temporary_head : Signal_List_Node_Ptr;
   Temp2 : Signal_List_Node_Ptr;
   tail  : Signal_List_Node_Ptr;

begin
   -- now we get the signal variable names

   ada.Text_IO.Put_Line("GETTING SIGNALS");

   Reset(Input, In_File);
   loop
      exit when Find_Next_Matching_Word(Input,"SIGNAL") = -1;
      Special_Get_Next_Word(Input,':',Word,Word_Length, Eof);
      ada.Text_IO.Put_Line("FIRST SIGNAL NAME IN
      LINE:"&Word(1..word_length));
      Temp2 := new Signal_List_Node;
      Temp2.Signal_Name := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
      Temp2.Next := null;

      temporary_head := temp2;
      tail := Temp2;
      loop
         Special_Get_Next_Word(Input,':',Word, Word_Length, Eof);
         exit when Word(1..Word_Length) = ":";
         Temp2 := new Signal_List_Node;
         Temp2.Signal_Name := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
         Temp2.Next := null;

         tail.Next := Temp2;
         Tail := Temp2;
      end loop;


      --Get_Next_Word(Input,' ',Word, Word_Length, Eof);
      Tail.Next := Signal_List;
      Signal_List := Temporary_Head;
```

```
   end loop;

end Signals;
-------------------------------------------------------------



-------------------------------------------------------------
procedure Create_Output_Column (output_list_Head_ptr : in
Output_List_Node_Ptr; Output_Column_List_Ptr : out Output_Column_Ptr)
is

   Temp6 : Output_List_Node_Ptr := null;
   Temp7 : Output_Column_Ptr    := null;
   Temp8 : Output_Column_Ptr    := null;

begin
   if(Output_List_Head_Ptr /= null) then
      -- create the Output_Column list
      Temp6 := Output_List_Head_Ptr;
      Temp7:= new Output_Column;
      Temp7.Output_Name := Temp6.Output_Name;
      Temp7.Output_Value_List_Head_Ptr := null;
      Temp7.Next := null;

      Temp8 := Temp7;
      Output_Column_List_Ptr := Temp7;

      loop
         Temp6 := Temp6.Next;
         exit when Temp6 = null;
         Temp7:= new Output_Column;
         Temp7.Output_Name := Temp6.Output_Name;
         Temp7.Output_Value_List_Head_Ptr := null;
         Temp7.Next := null;

         Temp8.Next := Temp7;
         Temp8 := Temp7;
      end loop;
   else
      Ada.Text_Io.Put_Line("created empty output column");
   end if;

end Create_Output_Column;
-------------------------------------------------------------



-------------------------------------------------------------
procedure Create_Signal_Column (signal_list_Head_ptr : in
signal_List_Node_Ptr; signal_Column_List_Ptr : out signal_Column_Ptr)
is

   Temp6 : signal_List_Node_Ptr := null;
   Temp7 : signal_Column_Ptr    := null;
   Temp8 : signal_Column_Ptr    := null;
```

```
begin
   if(signal_List_Head_Ptr /= null) then
      -- create the Output_Column list
      Temp6 := signal_List_Head_Ptr;
      Temp7:= new signal_Column;
      Temp7.signal_Name := Temp6.signal_Name;
      Temp7.signal_Value_List_Head_Ptr := null;
      Temp7.Next := null;

      Temp8 := Temp7;
      signal_Column_List_Ptr := Temp7;

      loop
         Temp6 := Temp6.Next;
         exit when Temp6 = null;
         Temp7:= new signal_Column;
         Temp7.signal_Name := Temp6.signal_Name;
         Temp7.signal_Value_List_Head_Ptr := null;
         Temp7.Next := null;

         Temp8.Next := Temp7;
         Temp8 := Temp7;
      end loop;
   else
      Ada.Text_Io.Put_Line("created empty signal column");
   end if;

end Create_Signal_Column;
-------------------------------------------------------------------


-------------------------------------------------------------------
procedure Add_Row (
   Output_List_Head_Ptr : Output_List_Node_Ptr;
   signal_List_Head_Ptr : signal_List_Node_Ptr;
   State_Variable :  Wide_String_Ptr;
   State_Name : Wide_String_Ptr;
   State_Variable_list : in out State_Variable_Ptr) is

   Temp1       : Row_Ptr := null;
   Temp2       : Row_Ptr := null;
   Row_head    : Row_Ptr;
   Temp : State_Variable_Ptr;
   Temp5 : Next_State_Node_Ptr := null;
   Temp9 : Output_Column_Ptr   := null;
   Temp10 : signal_Column_Ptr   := null;

begin

   Temp := State_Variable_List;
   loop
      if (Temp = null) then
            Ada.Text_Io.Put_Line("ERROR: ADD ROW FUNCTION: NO
            CORRESPONDING STATE VARIABLE FOUND!");
            return;
      end if;
```

```
         exit when
Compare_Words(Wide_String_To_String(Temp.Signal_Name.all,1),Wide_String
_To_String(State_Variable.all,1));
         Temp :=Temp.next;
      end loop;

      temp2 := temp.truth_table;
      loop
         exit when Temp2 = null;
         if
(Compare_Words(Wide_String_To_String(Temp2.Present_State.all,1),Wide_St
ring_To_String(State_name.all,1))) then
            return;
         end if;
         temp2 := temp2.next;
      end loop;

      Temp1 := new Row;
      Temp1.Present_State := State_Name;
      Temp1.Next_State_Head_Ptr := null;
      Create_Output_Column (output_list_Head_ptr, temp9);
      Temp1.Output_Column_List_Ptr := Temp9;
      Create_Signal_Column (signal_list_Head_ptr, temp10);
      Temp1.Signal_Column_List_Ptr := Temp10;

      Temp1.Next := Temp.Truth_Table;
      Temp.Truth_Table := Temp1;

   end Add_Row;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   procedure Create_State_Variable_Entry(State_Variable_list : in out
   State_Variable_ptr; Signal_Name : Wide_String_ptr) is

      temp : State_variable_ptr;

   begin

      temp := State_Variable_list;
      loop
         exit when Temp = null;
         if
         (Compare_Words(Wide_String_To_String(Temp.Signal_Name.All,1),
         Wide_String_To_String(Signal_Name.All,1))) then
            ada.Text_IO.Put_Line("TRIED TO ADD STATE VARIABLE TO
            GENERAL LIST, BUT STATE VARIABLE ALREADY EXISTS!");
            return;
         end if;
         temp := temp.next;
      end loop;

      Temp := new State_Variable_Node;
      Temp.Signal_Name := Signal_Name;
      Temp.next := State_variable_list;
      State_Variable_List := Temp;
```

```
      ada.Text_IO.Put_Line("ADDED STATE VARIABLE TO GENERAL LIST");
   end Create_State_Variable_Entry;
   ------------------------------------------------------------


end Table_Setup_Functions;
-------------------------------------------------------------------------

-------------------------------------------------




TABLE_WRITE.ADB
-------------------------------------------------------------------------

-------------------------------------------------

package body Table_Write_Functions is

   ------------------------------------------------------------------
   procedure Fill_In_Transition (
         State_Info: Signal_Name_Value_Node_Ptr;
         Next_State : in     Wide_String_Ptr;
         Conditions_List :    Conditional_Node_Ptr;
         State_Variable_list: State_Variable_ptr ) is

      Temp2          : Next_State_Node_Ptr    := null;
      Temp           : Row_Ptr;
      Input_List_Ptr : Conditional_Node_Ptr;
      Temp1 :  State_Variable_Ptr;
      Temp3          : Next_State_Node_Ptr    := null;

   begin

      Temp1 := State_Variable_List;
      loop
         if (Temp1 = null) then
               Ada.Text_Io.Put_Line("ERROR: Fill IN transition: NO
               CORRESPONDING STATE VARIABLE FOUND!");
               return;
         end if;
         exit when
         Compare_Words(Wide_String_To_String(Temp1.SIgnal_Name.all,1),
         Wide_String_To_String(State_info.SIgnal_name.all,1));
         Temp1 :=Temp1.next;
      end loop;


      Temp := Temp1.truth_table;
      loop
         exit when Temp = null;
         if
        (Compare_Words(Wide_String_To_String(Temp.Present_State.all,1),
        Wide_String_To_String(State_info.Signal_value.all,1))) then
            Temp2 := new Next_State_Node;
```

```
         Temp2.Next_State_Name := Next_State;
         Temp2.Conditions := Conditions_List;
         --      Conditions_List := null;
         --Temp2.priority := state_info.priority;
         Temp.Global_Priority_Counter :=
         Temp.Global_Priority_Counter + 1;
         Temp2.Priority := Temp.Global_Priority_Counter;
         Temp2.Next := null;

         Temp3 := Temp.Next_State_Head_Ptr;
         if(Temp3 = null) then
            Temp.Next_State_Head_Ptr := Temp2;
         else
            loop
               exit when Temp3.Next = null;
               Temp3 := Temp3.Next;
            end loop;
            Temp3.Next := Temp2;
         end if;
      end if;
      Temp := Temp.Next;
   end loop;

end Fill_In_Transition;
-------------------------------------------------------------------


-------------------------------------------------------------------
procedure Fill_In_Output_Value (
      State_Info: Signal_Name_Value_Node_Ptr;
      Output_Name    : in     Wide_String_Ptr;
      Output_Value : in Wide_String_Ptr;
      Conditions_List :    Conditional_Node_Ptr;
      State_Variable_list: State_Variable_ptr ) is

   Temp2          : Output_Value_Node_Ptr;
   Temp           : Row_Ptr;
   Input_List_Ptr : Conditional_Node_Ptr;
   Temp1 : Output_Column_Ptr;
   Temp3 :  State_Variable_Ptr;

begin

   Temp3 := State_Variable_List;
   loop
      if (Temp3 = null) then
            Ada.Text_Io.Put_Line("ERROR: Fill IN transition: NO
            CORRESPONDING STATE VARIABLE FOUND!");
            return;
      end if;
      exit when
     (Compare_Words(Wide_String_To_String(Temp3.Signal_Name.All,1),
      Wide_String_To_String(State_Info.SIgnal_name.All,1)));
      Temp3 :=Temp3.next;
   end loop;
```

```
      Temp := Temp3.truth_table;

      loop
         exit when Temp = null;
         if
         (Compare_Words(Wide_String_To_String(Temp.Present_State.All,1),
         Wide_String_To_String(State_Info.SIgnal_value.All,1))) then
            Temp1 := Temp.Output_Column_List_Ptr;
            loop
               exit when Temp1 = null;
               if
          (Compare_Words(Wide_String_To_String(Temp1.Output_name.All,1),
          Wide_String_To_String(Output_name.All,1))) then
                  Temp2 := new Output_Value_Node;
                  Temp2.Input_Combinations := Conditions_List;
-- we can do this, because this copy of conditional_List will no be
used-- anywhere else -- process_statement makes a separate copy
                  Temp2.Output_Value := Output_Value;
                  Temp2.next := Temp1.Output_Value_List_Head_Ptr;
                  Temp1.Output_Value_List_Head_Ptr := Temp2;
               end if;
               Temp1 := Temp1.Next;
            end loop;
         end if;
         Temp := Temp.next;
      end loop;

   end Fill_In_Output_Value;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   procedure Fill_In_Signal_Value (
         State_Info: Signal_Name_Value_Node_Ptr;
         Signal_Name    : in     Wide_String_Ptr;
         Signal_Value : in Wide_String_Ptr;
         Conditions_List :    Conditional_Node_Ptr;
         State_Variable_list: State_Variable_ptr) is

      Temp2           : Signal_Value_Node_Ptr;
      Temp            : Row_Ptr;
      Input_List_Ptr : Conditional_Node_Ptr;
      Temp1 : Signal_Column_Ptr;
      Temp3 :  State_Variable_Ptr;

   begin

      Temp3 := State_Variable_List;
      loop
         if (Temp3 = null) then
               Ada.Text_Io.Put_Line("ERROR: Fill IN transition: NO
               CORRESPONDING STATE VARIABLE FOUND!");
               return;
         end if;
         exit when
         (Compare_Words(Wide_String_To_String(Temp3.Signal_Name.All,1),
         Wide_String_To_String(State_Info.SIgnal_name.All,1)));
```

```
         Temp3 :=Temp3.next;
      end loop;


      Temp := Temp3.truth_table;

      loop
         exit when Temp = null;
         if
        (Compare_Words(Wide_String_To_String(Temp.Present_State.All,1),
        Wide_String_To_String(State_Info.SIgnal_value.All,1))) then
            Temp1 := Temp.Signal_Column_List_Ptr;
            loop
               exit when Temp1 = null;
               if
          (Compare_Words(Wide_String_To_String(Temp1.Signal_name.All,1),
               Wide_String_To_String(Signal_name.All,1))) then
                  Temp2 := new Signal_Value_Node;
                  Temp2.Input_Combinations := Conditions_List;
-- we can do this, because this copy of conditional_List will no be
used -- anywhere else -- process_statement makes a separate copy
                  Temp2.Signal_Value := Signal_Value;
                  Temp2.next := Temp1.Signal_Value_List_Head_Ptr;
                  Temp1.Signal_Value_List_Head_Ptr := Temp2;
               end if;
               Temp1 := Temp1.Next;
            end loop;
         end if;
         Temp := Temp.next;
      end loop;

   end Fill_In_Signal_Value;
   ------------------------------------------------------------------

-----------------------------------------------------------------------
-------------------------------------------------
```

TRUTH_TABLE_GRAPHER.ADB

```
-----------------------------------------------------------------------

-------------------------------------------------

package body Truth_Table_Grapher is


procedure Grapher (State_list : in out State_variable_ptr) is

   res : String( 1 .. 10 );
   Temp : Integer;
   N1 : Node;
   Var : wide_string_ptr;
```

```
   Temp1 : Row_Ptr;
   Temp2 : Next_State_Node_Ptr;
   Temp6       : State_variable_ptr;
   flag : boolean;

begin

   Nnodes  := 1;
   Nedges  := 1;

   flag := true;

   Ada.Numerics.Float_Random.Reset(Seed);

   Temp6  := State_list;
--   loop
--      exit when Temp6 = null;

      Temp1 := Temp6.truth_table;

      loop
         exit when Temp1 = null;
         if(temp6.start_state = null) then
            Temp := Addnode(Temp1.Present_State);

elsif(Compare_Words(Wide_String_To_String(temp6.start_state.all,1),Wide
_String_To_String(temp1.present_state.all,1))) then
            Temp := Addnode(Temp1.Present_State,null,false,false,True);
         else
            Temp := Addnode(Temp1.Present_State);
         end if;
         Temp1 := Temp1.next;
      end loop;

      Temp1 := Temp6.truth_table;

      loop
         exit when Temp1 = null;
         Temp2 := Temp1.Next_State_Head_Ptr;
         loop
            exit when Temp2 = null;
            Addedge
(Temp1.Present_State,Temp2.Next_State_Name,Double(600));
            Temp2 := Temp2.next;
         end loop;
         Temp1 := Temp1.next;
      end loop;

--      Temp6 := Temp6.next;
--   end loop;

if (flag) then
   for I in 1..Nnodes-1 loop
      Put(Nodes(I).Name.all&":");
      Put(float(Nodes(i).x));
      var :=  new Wide_String'(" ");Put_Line(Var.all);
      Put(float(Nodes(i).y));
```

```
      var :=  new Wide_String'(" ");put_Line(var.all);
   end loop;
end if;

   Relax(seed);

if (flag) then
   for I in 1..Nnodes-1 loop
      Put(Nodes(I).Name.all&":");
      Put(float(Nodes(i).x));
      var :=  new Wide_String'(" ");Put_Line(Var.all);
      Put(float(Nodes(i).y));
      Var :=  new Wide_String'(" ");Put_Line(Var.all);
   end loop;
end if;

   Temp6  := State_list;

--   loop
--      exit when Temp6 = null;

      Temp1 := Temp6.truth_table;

      loop
         exit when Temp1 = null;

Find_Location(Temp1.Present_State,Temp1.Location_X,Temp1.Location_Y);
         temp1 := temp1.next;
      end loop;

--      temp6 := temp6.next;
-- end loop;

ada.Text_IO.Put_Line("GRAPGHING OPTIMIZER DONE");

end Grapher;
-------------------------------------------------------------------------
-------------------------------------------------
```

GRAPHING4.ADB

```
-------------------------------------------------------------------------

-------------------------------------------------
```

- The code below was written by Carl Nehme at the dept of Aeronautics and Astronautics at MIT
- Copyright © 2004
- The code is a modified version of the Spring Embedder Algorithm and the original source code was taken from

```
*************************************************************************
* Copyright (c) 1994-1996 Sun Microsystems, Inc. All Rights Reserved.
```

```
 * @modified 96/04/24 Jim Hagen : changed stressColor
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL or COMMERCIAL purposes and
 * without fee is hereby granted.
 * Please refer to the file http://java.sun.com/copy_trademarks.html
 * for further important copyright and trademark information and to
 * http://java.sun.com/licensing.html for further important licensing
 * information for the Java (tm) Technology.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 ************************************************************************
```

- The code is used to the visual appearance of a graph by minimizing edge crossings
- The nodes are repositioned in such a way that there is very few edges crossing

```
package body graphing4 is



-----------------------------------------------------------------------
   function Itos (I : integer ) return String is
      Res : String (1 .. 40);
      J   : Integer;

   begin
      Put(Res, I);
      J := Res'Last;    -- Find end of the stupid leading spaces.
      loop
         exit when J < Res'First;
         exit when Res(J) = ' ';
         J := J - 1;
      end loop;
      return Res(J + 1 .. Res'Last);

   end Itos;


   -----------------------------------------------------------------


   -----------------------------------------------------------------
   function ftos (f : float ) return String is
      Res : String (1 .. 40);
      J   : Integer;

   begin
      Put(Res, f);
      J := Res'Last;    -- Find end of the stupid leading spaces.
      loop
```

```
      exit when J < Res'First;
      exit when Res(J) = ' ';
      J := J - 1;
   end loop;
   return Res(J + 1 .. Res'Last);

end ftos;
-------------------------------------------------------------------


-------------------------------------------------------------------
function Addnode (Name : in   Wide_String_Ptr; Invariant :
wide_string_ptr:= null; Urgent : boolean := false; Committed :
boolean := false; Initial : boolean := false) return integer is

   N : Node;

begin
   if (Initial) then
      N.X := 30.0;
      n.dx := double(0);
      N.Y := 30.0;
      n.dy := double(0);
   else
      N.X := Double(10) +
      Double(380)*Double(Ada.Numerics.Float_Random.Random(Seed));
      n.dx := double(0);
      N.Y := double(10) +
      double(380)*double(Ada.Numerics.Float_Random.Random(seed));
      N.Dy := Double(0);
   end if;
   N.Name := Name;
   N.Initial := Initial;
   N.urgent := Urgent;
   N.Committed := Committed;
   N.Invariant := Invariant;
   Nodes(Nnodes) := N;
   nnodes := nnodes + 1;
   return nnodes;
end addNode;
-------------------------------------------------------------------


-------------------------------------------------------------------
 function findNode (Name : in   Wide_String_Ptr) return integer is

      Var : wide_String_ptr;

 begin

      for I in 1..Nnodes-1 loop
         if (Nodes(I).Name.all = Name.all) then
            return I;
         end if;
      end  loop;

      return -1;
```

```
end findNode;
-----------------------------------------------------------------


-----------------------------------------------------------------
procedure addEdge (from : in   Wide_String_Ptr;
                   To : in Wide_String_Ptr;
                   len : double ) is

   e : Edge;

   begin
      E.From := Findnode(From);
      if (E.From = -1) then
         Put_line("Node not found:
         "&Wide_String_To_String(From.All,1));
         return;
      end if;
      E.To := Findnode(To);
      if (E.to = -1) then
         Put_line("Node not found:
         "&Wide_String_To_String(to.All,1));
         return;
      end if;
      e.len := len;
      edges(nedges) := e;
      nedges := nedges + 1;
end addEdge;
-----------------------------------------------------------------


-----------------------------------------------------------------
procedure relax (seed : in  Ada.Numerics.Float_Random.Generator) is

   E : Edge;
   Vx : double;
   Vy : double;
   Len : double;
   F : double;
   Dx : double;
   Dy : Double;
   n : node;
   N1 : Node;
   N2 : Node;
   Dlen : double;
   M: Float;
   C1: Double := 2.0;
   C2: Double := 1.0;
   C3: Double := 2.0;
   C4: Double := 0.1;
   Iterations: integer := 3000;
   Repulsive_Force : Double;

begin

 for u in 1..4 loop
```

```
      if(U = 2) then
         C3 := 500.0;
      end if;

      if(U = 3) then
         C3 := 4000.0;
      end if;

      if(U = 4) then
         C3 := 8000.0;
      end if;

--      if(U = 5) then
--        C3 := 100000.0;
--      end if;

      for K in 1..Iterations loop

         for I in 1..Nnodes-1 loop
            N1 := Nodes(I);
            Dx := double(0);
            Dy := double(0);

            for J in I+1..Nnodes-1 loop
                  N2 := Nodes(J);
                  Vx := N2.X - N1.X;
                  Vy := N2.Y - N1.Y;
                  Len := Vx * Vx + Vy * Vy;

                  if (Len = double(0)) then
                     len := 1.0;
                  end if;

                  Dlen := Sqrt(len);
                  Repulsive_Force := C3/Len;
                  Nodes(I).Dx   := Nodes(I).Dx - Repulsive_Force*Vx/Dlen;
                  Nodes(I).Dy   := Nodes(I).Dy - Repulsive_Force*Vy/Dlen;
                  Nodes(J).Dx   := Nodes(J).Dx + Repulsive_Force*Vx/Dlen;
                  Nodes(J).Dy   := Nodes(J).Dy + Repulsive_Force*Vy/Dlen;

            end loop;
         end loop;

         for I in 1..Nedges-1 loop
            E := Edges(I);
            Vx := Nodes(E.To).X - Nodes(E.From).X;
            Vy := Nodes(E.To).Y - Nodes(E.From).Y;
            Len := Vx * Vx + Vy * Vy;
            if (Len = double(0)) then
                     len := 1.0;
            end if;
            dlen := sqrt(len);
            F := (C1*Log(dlen/c2)) / log(10.0) ;

            Dx := (F * Vx)/DLEN;
            Dy := (F * VY)/DLEN;
```

```
            Nodes(E.To).Dx := Nodes(E.To).Dx - Dx;
            Nodes(E.To).Dy := Nodes(E.To).Dy - Dy;
            Nodes(E.From).Dx := Nodes(E.From).Dx + Dx;
            Nodes(E.From).Dy := Nodes(E.From).Dy + Dy;
         end loop;

         for I in 1..Nnodes-1 loop
            if(not Nodes(I).initial) then
               Nodes(I).X := double'max(Nodes(I).X +
               C4*Nodes(I).Dx,50.0);
               Nodes(I).Y := double'max(Nodes(I).Y +
               C4*Nodes(I).Dy,50.0);
            end if;
            Nodes(I).Dx := 0.0;
            Nodes(I).Dy := 0.0;
         end loop;


         end loop;

      end loop;

   end relax;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   procedure Find_location(name : wide_string_ptr; X : out integer; Y :
   out integer) is

   N1 : Node;

   begin

     for I in 1..Nnodes-1 loop
         N1 := Nodes(I);
         if ((N1.Name.All = Name.All)) then
            X := Integer(N1.X);
            Y := integer(N1.Y);
            return;
         end if;
      end loop;

   end Find_location;
------------------------------------------------------------------


------------------------------------------------------------------
-------------------------------------------------
```

```
PARSER_ADVANCED6.ADB
-------------------------------------------------------------------

-------------------------------------------------

-- comment: when comparing wide_string pointers for equality, one needs
to use the ".all" extension after the name of the wide_string pointer
to get
-- a proper comparison, ADA will allow the comparison of wide_string
pointers without the use of the ".all" extension, however this will
give an erroneous result

-- when copying data structures, a simple assignment statement suffices,
no explicit copying of individual componenents needed


with Ada.Characters;
use Ada.Characters;
with Ada.Sequential_Io;
use Ada;
with Ada.Integer_Text_Io;
use Ada.Integer_Text_Io;
with Ada.Numerics.Float_Random;
use Ada.Numerics.Float_Random;
with Ada.Numerics.Aux;
use Ada.Numerics.Aux;
with Ada.Numerics.Generic_Elementary_Functions;
with Ada.Strings.Wide_Fixed;
use Ada.Strings.Wide_Fixed;
with Ada.Strings.Wide_Unbounded;
use Ada.Strings.Wide_Unbounded;
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;
with Ada.Wide_Text_Io;
use Ada.Wide_Text_Io;
with Ada.Float_Text_Io;
use Ada.Float_Text_Io;
with Ada.Text_Io;
use Ada.Text_Io;
with System.Wch_Wts;
use System.Wch_Wts;
with Ada.Characters.Handling; use Ada.Characters.Handling;
--with gtk.main, gtk.window;

--package my_float is new ada.text_io(float );

-- import of other packages written for the purpose of this program
with parser_types; use parser_Types;
with Parsing_Functions; use Parsing_Functions;
with output_functions; use output_functions;
with Table_Setup_Functions; use Table_Setup_Functions;
with table_Write_Functions; use table_Write_Functions;
with graphing4; use graphing4;
with conditional_list_handlers; use conditional_list_handlers;
with statement_handlers; use statement_handlers;
with truth_table_grapher; use truth_table_grapher;
```

```
procedure Parser is

   File_Name        :  unbounded_string;
   Output_File_Name : constant String                := "out.xml";
   Template         : constant String                := "template.xml";
   Header1          : constant String                := "<?xml
version=""1.0"" encoding=""UTF-8""?><!DOCTYPE nta PUBLIC ""-//Uppaal
Team//DTD Flat System 1.0//EN""
""http://www.docs.uu.se/docs/rtmv/uppaal/xml/flat-1_0.dtd"">";
   Header2          : constant String                :=
"<nta><declaration>";
   Header2_1        : constant String                :=
"</declaration>";
   Header3          : constant String                := "<template>";
   Header4          : constant String                := "<parameter
x=""176"" y=""16"">const pid</parameter><declaration>";
   Header5          : constant String                := "int
clk;</declaration>";
   Footer1          : constant String                := "</template>";
   Footer2          : constant String                :=
"<instantiation>";
   Footer3          : constant String                :=
"</instantiation><system>system ";
   Footer4          : constant String                :=
"</system></nta>";
   Input            :          Ada.Text_Io.File_Type;
   Output           :          Ada.Text_Io.File_Type;
   Word             :          Lines;
   Word_Length      :          Integer;
   Eof              :          Boolean;
   Tab              :          Character             := Character'Val
(9);

   zoom_factor      : float;

   Input_List_Head_Ptr  : Input_List_Node_Ptr  := null;
   Output_List_Head_Ptr : Output_List_Node_Ptr := null;
   Input_List_Length    : Integer;
   Signal_List          : Signal_List_Node_Ptr := null;

   signal_flag : boolean := false;
   output_Flag : Boolean := False;

     function Find_State_Variable (
         State_variable_list  : State_variable_Ptr;
         State_variable_Name : Wide_String_Ptr )
     return State_variable_Ptr;



   ------------------------------------------------------------------
   -- UNUSED!!!!
   ------------------------------------------------------------------
   procedure COPY_Input_Combinations_List (
         Input_List     : in     Input_Value_Node_Ptr;
         Input_List_Ptr :    out Input_Value_Node_Ptr  ) is
```

```
   Temp  : Input_Value_Node_Ptr;
   Temp1 : Input_Value_Node_Ptr;
   Temp2 : Input_Value_Node_Ptr;

begin

   Temp2 :=null;
   Temp  := Input_List;

   loop
      exit when Temp = null;
      Temp1 := new Input_Value_Node;
      Temp1.Input_Name := Temp.Input_Name;
      Temp1.Input_Value := Temp.Input_Value;
      Temp1.Next := Temp2;
      Temp2 := Temp1;
      Temp := Temp.Next;
   end loop;

   Input_List_Ptr := Temp2;

end COPY_Input_Combinations_List;
-------------------------------------------------------------------


-------------------------------------------------------------------
procedure Ignore_Spaces (
      Eof :    out Boolean ) is

   C         : Character;
   Endofline : Boolean   := False;

begin
   Eof := False;
   loop
      if End_Of_File(Input) then
         Eof := True;
         return;
      elsif End_Of_Line(Input) then
         Skip_Line(Input);
      else
         Look_Ahead(Input, C, Endofline);
         exit when (C /= ' ') and (C /= Tab);
         Get(Input, C);
      end if;
   end loop;

end Ignore_Spaces;
-------------------------------------------------------------------
```

```
   ------------------------------------------------------------------
   procedure Find_Location (
         Row_head   : Row_Ptr;
         State_Name : in     Wide_String_Ptr;
         X          :    out Integer;
         Y          :    out Integer          ) is

      Temp : Row_Ptr := null;

   begin

      Temp := Row_Head;
      loop
         exit when
Compare_Words(Wide_String_To_String(Temp.Present_State.all,1),Wide_Stri
ng_To_String(State_name.all,1));
         Temp := Temp.Next;
         if (Temp = null) then
            ada.Text_IO.Put_Line("ERROR in function Find_Location :
State """&Wide_String_To_String(State_name.all,1)&""" not found when
searching table!");
         end if;
      end loop;

      X := Temp.Location_X;
      Y := Temp.Location_Y;


   end Find_Location;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   function cond_equality (
      Output_cond_List : conditional_node_ptr;
      conditional_list : conditional_node_ptr          ) return boolean
is

      Temp : conditional_node_ptr;
      Temp1 : conditional_node_ptr;

   begin

      if(Output_Cond_List = null) then
         return True;
      end if;

      Temp := output_cond_list;
      loop
         exit when Temp = null;
         Temp1 := conditional_list;
         loop
            if (Temp1 = null) then
               return False;
            end if;
            if
(Compare_Words(Wide_String_To_String(Temp.name.all,1),Wide_String_To_St
```

```
ring(temp1.name.all,1)) and
Compare_Words(Wide_String_To_String(Temp.value.all,1),Wide_String_To_St
ring(temp1.value.all,1))) then
                exit;
             end if;
             Temp1 := Temp1.Next;
          end loop;
          Temp := Temp.Next;
       end loop;

 --      Temp := conditional_list;
 --      loop
 --         exit when Temp = null;
 --         Temp1 := output_cond_list;
 --         loop
 --            if (Temp1 = null) then
 --               return False;
 --            end if;
 --            if
(Compare_Words(Wide_String_To_String(Temp.name.all,1),Wide_String_To_St
ring(temp1.name.all,1)) and
Compare_Words(Wide_String_To_String(Temp.value.all,1),Wide_String_To_St
ring(temp1.value.all,1))) then
 --               exit;
 --            end if;
 --            Temp1 := Temp1.Next;
 --         end loop;
 --         Temp := Temp.Next;
 --      end loop;

       return true;

   end cond_equality;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   procedure Find_output (
      Output_Value_List_Head_Ptr : Output_Value_Node_Ptr;
      Conditional_List : Conditional_Node_Ptr;
      Output_Value_List: in out Output_Value_Node_Ptr;
      temp_wide_string: out wide_string_ptr) is

      Temp : output_value_node_ptr;
      Temp_List : Conditional_Node_Ptr;
      output_value_ptr : output_value_node_ptr;

   begin

      temp_wide_string := null;
      Temp := output_value_list_head_ptr;
      loop
         exit when Temp = null;
         if (temp.output_value /= null) then
            if  (cond_equality(Temp.input_Combinations,conditional_list))
then
               temp_wide_string := Temp.Output_Value;
```

```
            else
               Copy_Conditional_List
(temp.input_combinations,temp_list);
               Output_Value_ptr := new Output_Value_Node;
               Output_Value_Ptr.Input_Combinations := Temp_List;
               Output_Value_Ptr.output_value := Temp.output_value;
               Output_Value_Ptr.Next := Output_Value_List;
               Output_Value_List := Output_Value_Ptr;
            end if;
         end if;
         Temp := Temp.Next;
      end loop;


   end Find_output;
   ----------------------------------------------------------------------

   ----------------------------------------------------------------------
   procedure Find_signal (
      signal_Value_List_Head_Ptr : signal_Value_Node_Ptr;
      Conditional_List : Conditional_Node_Ptr;
      signal_Value_List: in out signal_Value_Node_Ptr;
      temp_wide_string: out wide_string_ptr) is

      Temp : signal_value_node_ptr;
      Temp_List : Conditional_Node_Ptr;
      signal_value_ptr : signal_value_node_ptr;

   begin

      temp_wide_string := null;
      Temp := signal_value_list_head_ptr;
      loop
         exit when Temp = null;
         if (Temp.Signal_Value /= null) then
            if  (cond_equality(Temp.input_Combinations,conditional_list))
then
               Temp_Wide_String := Temp.Signal_Value;
            else
               Copy_Conditional_List
(temp.input_combinations,temp_list);
               signal_Value_ptr := new signal_Value_Node;
               signal_Value_Ptr.Input_Combinations := Temp_List;
               signal_Value_Ptr.signal_value := Temp.signal_value;
               signal_Value_Ptr.Next := signal_Value_List;
               signal_Value_List := signal_Value_Ptr;
            end if;
         end if;
         Temp := Temp.Next;
      end loop;

   end Find_signal;
   ----------------------------------------------------------------------

   ----------------------------------------------------------------------
   procedure Calculate_Location (
         X       : in     Integer;
```

```
      X1       : in     Integer;
      Y        : in     Integer;
      Y1       : in     Integer;
      X_Coord :    out Integer;
      Y_Coord :    out Integer  ) is

begin

   if ( X = X1 and Y = Y1) then
      X_Coord := X ;
      Y_Coord := Y - 50;

   elsif ((X = X1) and (Y < Y1)) then
      X_Coord := X;
      Y_Coord := (Y1-Y)/2 + Y;

   elsif ((X < X1) and (Y = Y1)) then
      X_Coord := (X1-X)/2 + X;
      Y_Coord := Y;

   elsif (X = X1 and Y > Y1) then
      X_Coord := X;
      Y_Coord := (Y-Y1)/2 + Y1;

   elsif (X > X1 and Y = Y1) then
      X_Coord := (X-X1)/2 + X1;
      Y_Coord := Y;

   elsif ((X < X1) and (Y < Y1)) then
      X_Coord := (X1-X)/2 + X;
      Y_Coord := (Y1-Y)/2 + Y;

   elsif ((X < X1) and (Y > Y1)) then
      X_Coord := (X1-X)/2 + X;
      Y_Coord := (Y-Y1)/2 + Y1;

   elsif (X > X1 and Y < Y1) then
      X_Coord := (X-X1)/2 + X1;
      Y_Coord := (Y1-Y)/2 + Y;

   elsif (X > X1 and Y > Y1) then
      X_Coord := (X-X1)/2 + X1;
      Y_Coord := (Y-Y1)/2 + Y1;
   end if;


end Calculate_Location;
-------------------------------------------------------------------
-------------------------------------------------------------------
function Addnail (
   Truth_Table : Row_ptr;
   Present_State : wide_String_ptr;
   Next_State_Name: wide_String_ptr) return boolean is

   Temp : Row_ptr;
   Temp1 : Next_State_Node_Ptr;
```

```
  begin

      if
(Compare_Words(Wide_String_To_String(Next_State_Name.all,1),Wide_String
_To_String(Present_State.all,1))) then
        return False;
      end if;

      temp := truth_table;
      loop
        exit when Temp = null;
        if
(Compare_Words(Wide_String_To_String(Next_State_Name.all,1),Wide_String
_To_String(Temp.Present_State.all,1))) then
           Temp1 := Temp.Next_State_Head_Ptr;
            loop
              exit when Temp1 = null;
              if
(Compare_Words(Wide_String_To_String(Present_State.all,1),Wide_String_T
o_String(Temp1.NExt_State_Name.all,1))) then
                 return True;
              end if;
              Temp1 := Temp1.Next;
            end loop;
            return false;
         end if;
         Temp := Temp.Next;
      end loop;

   end AddNail;
  ------------------------------------------------------------------------


   ------------------------------------------------------------------
  procedure Find_Guard_Location(
       Truth_Table : Row_Ptr;
       Present_State : Wide_String_Ptr;
       Next_State_Name : Wide_String_Ptr;
       X : Integer;
       Y : Integer;
       Guard_Location_X : out Integer;
       Guard_Location_Y : out Integer;
       Nail_Flag : out Boolean) is

     X_Diff : Integer;
     Y_Diff : Integer;
     X_Coord      : Integer;
     Y_Coord      : Integer;
     X_Next_State : Integer;
     Y_Next_State : Integer;

   begin

Find_Location(truth_table,Next_State_Name,X_Next_STate,Y_Next_STate);
      Calculate_Location(X_Next_STate,X,Y_Next_STate,Y,X_Coord,Y_Coord);
-- for normal point
      X_Diff := abs(X_Next_STate-X);
```

```
      Y_Diff := abs(Y_Next_STate-Y);

      if(Addnail(Truth_Table,Present_State,Next_State_Name)) then
         Nail_Flag := True;
         if(Wide_String_To_String(Next_State_Name.All,1) >
Wide_String_To_String(Present_State.All,1)) then
            Guard_Location_X := X_Coord-(Y_Diff/3);
               if(x_diff-Y_diff>50) then
                  Guard_Location_Y := Y_Coord-(X_Diff/3);
               else
                  Guard_Location_Y := Y_Coord;
               end if;
          else
             guard_Location_X := X_Coord+(Y_Diff/3);
             if(x_diff-Y_diff>50) then
                guard_Location_Y := Y_Coord+(X_Diff/3);
             else
                Guard_Location_Y := Y_Coord;
             end if;
         end if;
      else
         Nail_flag := false;
         Guard_Location_X := X_Coord;
         Guard_Location_Y := Y_Coord;
      end if;
   end Find_Guard_Location;
   ------------------------------------------------------------------

   ------------------------------------------------------------------
   function Opposite (Operator: wide_String_ptr) return String is

   begin

      if (Compare_Words(Wide_String_To_String(Operator.all,1),"="))
then
         return "!=";
      elsif (Compare_Words(Wide_String_To_String(Operator.all,1),">"))
then
         return "&lt;=";
      elsif (Compare_Words(Wide_String_To_String(Operator.all,1),">="))
then
         return "&lt;";
      elsif (Compare_Words(Wide_String_To_String(Operator.all,1),"<"))
then
         return "&gt;=";
      elsif (Compare_Words(Wide_String_To_String(Operator.all,1),"<="))
then
         return "&gt;";
      end if;
   end Opposite;
   ------------------------------------------------------------------

   function LT_GT_VALUE (Operator: wide_String_ptr;name:
wide_String_ptr;value: wide_String_ptr) return String is

   begin
      -- if(Operator != Null)
```

```
      --    Put(Output,Wide_String_To_String(Name.all,1));
      --    Put(Output,Wide_String_To_String(value.all,1));
      --    Put(Output,Wide_String_To_String(operator.all,1));
         if (Compare_Words(Wide_String_To_String(Operator.all,1),"="))
then
            return "==";
         elsif
(Compare_Words(Wide_String_To_String(Operator.all,1),">")) then
            return "&gt;";
         elsif
(Compare_Words(Wide_String_To_String(Operator.all,1),">=")) then
            return "&gt;=";
         elsif
(Compare_Words(Wide_String_To_String(Operator.all,1),"<")) then
            return "&lt;";
         elsif
(Compare_Words(Wide_String_To_String(Operator.all,1),"<=")) then
            return "&lt;=";
         elsif
(Compare_Words(Wide_String_To_String(Operator.all,1),"/=")) then
            return "/=";
         end if;
      --else
        -- return null;
   end LT_GT_VALUE;
   ------------------------------------------------------------------

   ------------------------------------------------------------------
   procedure Fill_In_File is
      Temp         : Row_Ptr                 := null;
      Temp10         : Row_Ptr                 := null;
      Temp1        : Next_State_Node_Ptr     := null;
      Temp2        : Output_Column_Ptr       := null;
      Temp3        : Output_Value_Node_Ptr   := null;
      Temp6        : State_Variable_Ptr;
      Counter      : Integer                 := 1;
      Increment    : Integer                 := 300;
      X            : Integer                 := 300;
      Y            : Integer                 := 50;

      Label_X      : Integer;
      Label_Y      : Integer;
      Counter_Str  : String (1 .. 10);
      Temp4        : Conditional_Node_Ptr;
      Temp5        : Input_Value_Node;
      out_temp      : output_column_ptr;
      Out_Temp1      : Output_Column_Ptr;
      Out_Temp2      : Output_Column_Ptr;
      out_temp_list : Output_Column_Ptr;
      temp_wide_string : wide_string_ptr;
      Output_Value_List:  Output_Value_Node_Ptr;
      Signal_temp      : signal_column_ptr;
      signal_Temp1      : signal_Column_Ptr;
      signal_Temp2      : signal_Column_Ptr;
      signal_temp_list : signal_Column_Ptr;
      signal_Value_List:  signal_Value_Node_Ptr;
      flag : boolean;
```

```
      Nail_Flag : Boolean;
      Guard_location_x : integer;
      Guard_Location_Y : Integer;
      line_length_flag : integer;
      Temp7          : Input_List_Node_Ptr;
      Temp8          : Output_List_Node_Ptr;
      Temp9          : Signal_List_Node_Ptr;

   begin
      Put(Output,Header1);
      Put(Output,Header2);
      Temp7 := Input_List_head_ptr;
      loop
         exit when Temp7 = null;
         Put(Output,Wide_String_To_String(" int
"&Temp7.Input_Name.all,1)&";");
         Temp7 := Temp7.Next;
      end loop;

      Temp8 := Output_List_head_ptr;
      loop
         exit when Temp8 = null;
         Put(Output,Wide_String_To_String(" int
"&Temp8.Output_Name.all,1)&";");
         Temp8 := Temp8.Next;
      end loop;

      Temp9 := Signal_List;
      loop
         exit when Temp9 = null;
         flag := false;
         Temp6 := State_Variable_List;
         loop
            exit when (Temp6 = null);
            if
(Compare_Words(Wide_String_To_String(Temp6.Signal_name.all,1),Wide_Stri
ng_To_String(Temp9.Signal_Name.all,1))) then
               flag := true;
            end if;
            Temp6 := Temp6.next;
         end loop;
         if (not (Flag)) then
            Put(Output,Wide_String_To_String(" int
"&Temp9.Signal_Name.all,1)&";");
         end if;
         Temp9 := Temp9.Next;
      end loop;
      Put(Output,Header2_1);

      Temp6 := State_Variable_List;
      loop
         exit when (Temp6 = null);
         Temp := Temp6.truth_table;

         Put(Output,Header3);
         Put(Output,"<name x=""32""
y=""16"">"&Wide_String_To_String(Temp6.Signal_NAme.all,1)&"</name>");
```

```
         Put(Output,Header4);
         -- put local variables here
         Put(Output,Header5);

         grapher(Temp6);
         Temp10 := Temp6.Truth_Table;
         loop
            exit when Temp10 = null;
            Temp10.Location_X :=
Integer(Double(Temp10.Location_X)*double(zoom_factor));
            Temp10.Location_Y :=
Integer(Double(Temp10.Location_Y)*double(zoom_factor));
            Temp10 := Temp10.Next;
         end loop;

        loop
            exit when Temp = null;

            X := Temp.Location_X;
            Y := Temp.Location_Y;

            Label_X := X + 20;
            Label_Y := Y - 20;

            ---  PUTTING STATE NAME
            Put(Output,"<location
id=""id_"&Wide_String_To_String(Temp6.Signal_Name.All,1)&"_"&Wide_Strin
g_To_String(Temp.Present_State.All,1)&""" x="""&Itos(X)&"""
y="""&Itos(Y)&""">");
            Put(Output,"<name x="""&Itos(Label_X)&"""
y="""&Itos(Label_Y)&""">");
            Put(Output,Wide_String_To_String(Temp.Present_State.all,1));
            Put(Output,"</name></location>");

            Temp := Temp.Next;
         end loop;

         if (Temp6.start_State /= null) then
            Put(Output,"<init
ref=""id_"&Wide_String_To_String(Temp6.Signal_Name.All,1)&"_"&Wide_Stri
ng_To_String(Temp6.Start_State.All,1)&"""/>");
         end if;

         Temp := Temp6.truth_table;

         loop
            exit when Temp = null;

            X := Temp.Location_X;
            Y := Temp.Location_Y;

            --- PUTTING ARROWS TO NEXT STATE
            Temp1 := Temp.Next_State_Head_Ptr;
            Out_Temp := Temp.Output_Column_List_Ptr;
            Signal_Temp := Temp.Signal_Column_List_Ptr;
            loop
               exit when Temp1 = null;
```

```
                Put(Output,"<transition>");
                Put(Output,"<source
ref=""id_"&Wide_String_To_String(Temp6.Signal_name.all,1)&"_"&Wide_Stri
ng_To_String(Temp.Present_State.all,1)&"""/>");
                Put(Output,"<target
ref=""id_"&Wide_String_To_String(Temp6.Signal_name.all,1)&"_"&Wide_Stri
ng_To_String(Temp1.Next_State_Name.all,1)&"""/>");

                Ada.Text_Io.Put_Line("GOING TO LOOK FOR
"""&Wide_String_To_String(Temp1.Next_State_Name.All,1)&""" in state
Table for """&Wide_String_To_String(Temp6.signal_name.all,1)&""""");



                --- PUTTING GUARD ON ARROW

Find_Guard_Location(Temp6.truth_table,Temp.present_State,Temp1.next_sta
te_name,X,Y,Guard_Location_X,Guard_Location_Y,nail_flag);
                Put(Output,"<label kind=""guard""
x="""&Itos(Guard_Location_X)&""" y="""&Itos(Guard_Location_Y)&""">");

              --  if(temp1.priority > 1 or ((Temp1.priority = 1) and
(temp1.next /= null))) then
              --      Put(Output,"--"&Itos(Temp1.Priority)&"--");
              --  end if;
                Flag := True;
                Temp4 := Temp1.Conditions;
                loop
                   exit when Temp4 = null;
                   if (Temp4.Name.All(1) /= '*') then
                      if(Flag) then
                         Flag := False;
                      else
                         Put(Output,", ");
                      end if;

Put(Output,Wide_String_To_String(Temp4.Name.all,1)&LT_GT_VALUE(Temp4.Op
erator,Temp4.name,Temp4.value)&Wide_String_To_String(Temp4.Value.all,1)
);
                   else
                      if(Flag) then
                         Flag := False;
                      else
                         Put(Output,", ");
                      end if;

Put(Output,Wide_String_To_String(Temp4.Name.all(2..temp4.name'length),1
)&Opposite(Temp4.Operator)&Wide_String_To_String(Temp4.Value.all,1));
                   end if;
                   Temp4 := Temp4.Next;
                end loop;
                Put(Output,"</label>");

             if(Nail_Flag) then

Ada.Text_Io.Put_Line(Wide_String_To_String(Temp.Present_State.All,1)&">
```

```
"&Itos(X)&":"&Itos(Y)&"
"&Wide_String_To_String(Temp1.Next_State_Name.All,1)&">");

Ada.Text_Io.Put_Line(Wide_String_To_String(Temp.Present_State.All,1)&"&
"&Wide_String_To_String(Temp1.Next_State_Name.All,1)&">");
                Put(Output,"<nail x="""&Itos(Guard_location_X)&"""
y="""&Itos(Guard_location_Y)&""" />");
            end if;


            if(Output_Flag) then
               Put(Output,"<label kind=""assignment""
x="""&Itos(Guard_Location_X)&"""  y="""&Itos(Guard_Location_Y+25)&""">");
               Out_Temp1 := Out_Temp;
               Out_Temp_List := null;
               Flag := True;
               line_length_flag := 0;
               loop
                  exit when Out_Temp1 = null;
                  output_value_list := null;
                  Find_Output
(Out_Temp1.Output_Value_List_Head_Ptr,temp1.conditions,output_value_lis
t,temp_wide_String);
                  if (Temp_Wide_String /= null) then
                     if(Flag) then
                        Flag := False;
                     else
                        Put(Output,", ");
                     end if;

Put(Output,Wide_String_To_String(Out_Temp1.Output_Name.All,1)&":=
"&Wide_String_To_String(Temp_Wide_String.All,1));
                  else
                     if (output_value_list /= null) then
                        out_temp2 := new output_column;
                        Out_Temp2.Output_Name := Out_Temp1.Output_Name;
                        Out_Temp2.Output_Value_List_Head_Ptr :=
Output_Value_List;
                        Out_Temp2.Next := Out_Temp_List;
                        Out_Temp_List := Out_Temp2;
                      end if;
                  end if;
                  Out_Temp1 := Out_Temp1.Next;
               end loop;
   --            Put(Output,"</label>");
            end if;

            if(Signal_Flag) then
             --  Put(Output,"<label kind=""synchronisation""
x="""&Itos(Guard_Location_X)&"""  y="""&Itos(Guard_Location_Y+50)&""">");
               Signal_Temp1 := Signal_Temp;
               Signal_temp_list := null;
               Flag := True;
               line_length_flag := 0;
               loop
                  exit when Signal_Temp1 = null;
                  Signal_Value_List := null;
```

```
                  Find_SIgnal
(Signal_Temp1.Signal_Value_List_Head_Ptr,temp1.conditions,Signal_value_
list,temp_wide_String);
                  if (Temp_Wide_String /= null) then
                     if(Flag) then
                        Flag := False;
                     else
                        Put(Output,", ");
                     end if;

Put(Output,Wide_String_To_String(Signal_Temp1.Signal_Name.All,1)&":=
"&Wide_String_To_String(Temp_Wide_String.All,1)&"  ");
                  else
                     if (Signal_value_list /= null) then
                        Signal_temp2 := new Signal_column;
                        Signal_Temp2.signal_Name :=
Signal_Temp1.Signal_Name;
                        SIgnal_Temp2.signal_Value_List_Head_Ptr :=
Signal_Value_List;
                        Signal_Temp2.Next := Signal_Temp_List;
                        Signal_Temp_List := Signal_Temp2;
                      end if;
                  end if;
                  Signal_Temp1 := Signal_Temp1.Next;
               end loop;
               Put(Output,"</label>");
            end if;


            Put(Output,"</transition>");
            Temp1 := Temp1.Next;
         end loop;

--             if (out_temp_list /= null) then
--
--                Put(Output,"<transition>");
--                Put(Output,"<source
ref=""id_"&Wide_String_To_String(Temp6.Signal_name.all,1)&"_"&Wide_Stri
ng_To_String(Temp.Present_State.all,1)&"""/>");
--                Put(Output,"<target
ref=""id_"&Wide_String_To_String(Temp6.Signal_name.all,1)&"_"&Wide_Stri
ng_To_String(Temp.Present_State.all,1)&"""/>");
--
Calculate_Location(Temp.Location_X,Temp.Location_X,Temp.Location_Y,Temp
.Location_Y,X_Coord,Y_Coord);
--                Put(Output,"<label kind=""guard""
x="""&Itos(X_Coord)&""" y="""&Itos(Y_Coord)&""">");
--                Out_Temp1 := Out_Temp_list;
--                loop
--                   exit when Out_Temp1 = null;
--                   Output_Value_List :=
Out_Temp1.Output_Value_List_Head_Ptr;
--                   loop
--                      exit when output_value_list = null;
--
Put_line(Output,Wide_String_To_String(Out_Temp1.Output_Name.All,1)&"="&
Wide_String_To_String(output_value_list.output_value.All,1)&"  ");
```

```
--                      Output_Value_List := Output_Value_List.Next;
--                   end loop;
--                   Out_Temp1 := Out_Temp1.Next;
--                end loop;
--                Put(Output,"</label>");
--                Put(Output,"</transition>");
--             end if;
            Temp := Temp.Next;
         end loop;

         Put(Output,Footer1);
         Temp6 :=Temp6.next;
      end loop;

      Put(Output,Footer2);
      Temp6 := State_Variable_List;
      loop
         exit when (Temp6 = null);
         Put(Output,Wide_String_To_String(Temp6.Signal_Name.all,1)&"1:=
"&Wide_String_To_String(Temp6.Signal_Name.all,1)&"(1);");
         Temp6 := Temp6.Next;
      end loop;
      Put(Output,Footer3);
      Temp6 := State_Variable_List;
      loop
         Put(Output,Wide_String_To_String(Temp6.Signal_Name.all,1)&"1");
         Temp6 := Temp6.Next;
         exit when (Temp6 = null);
         Put(Output,",");
      end loop;
      Put(Output,";");
      Put(Output,Footer4);

   end Fill_In_File;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   procedure Set_Initial_State(State_Variable_Name : Wide_String_Ptr)
is

      temp_int : integer;
      first_word : wide_String_ptr;
      S_Type : Statement_Type;
      value : wide_string_ptr;
      Temp : State_variable_Ptr;

   begin
      if(State_Variable_Name = null) then
         Ada.Text_Io.Put_Line("Reset state cannot be found for a null
state!");
         return;
      end if;

      Ada.Text_Io.Put_Line("Set_initial_State:
"&Wide_String_To_String(State_Variable_Name.All,1));
      Reset(Input, In_File);
```

```
      loop
         First_Word := Next_Word(Input,' ');
         if (First_Word = null) then
            ada.Text_IO.Put_Line("NO INITIAL STATE FOUND FOR:
"&Wide_String_To_String(State_Variable_Name.All,1));
            return;
         end if;

         if
(Compare_Words(Wide_String_To_String(First_Word.all,1),"CASE")) then
            find_end_for(Input,"case");

elsif(Compare_Words(Wide_String_To_String(First_Word.All,1),"IF")) then
            temp_int := Find_Next_Matching_Word(Input,"THEN");

elsif(Compare_Words(Wide_String_To_String(First_Word.all,1),"ELSIF"))
then
            temp_int := Find_Next_Matching_Word(Input,"THEN");

elsif(Compare_Words(Wide_String_To_String(First_Word.all,1),"ELSE"))
then
            temp_int := Find_Next_Matching_Word(Input,"THEN");

elsif(Compare_Words(Wide_String_To_String(First_Word.All,1),"END"))
then
            Get_until(Input,';',word,word_length,eof);
         else
                  Get_until(Input,';',word,word_length,eof);
                  Value := new
Wide_String'(characters.handling.To_Wide_String(Word(1..Word_Length)));
-- we are getting assignment Value

if(Compare_Words(Wide_String_To_String(first_word.all,1),Wide_String_To
_String(state_variable_name.all,1))) then
                     Ada.Text_Io.Put_Line("FOUND INITIAL,
"&Wide_String_To_String(First_Word.All,1)&" =
"&Wide_String_To_String(Value.All,1));
                     Temp := Find_State_Variable
(State_Variable_List,State_Variable_Name);
                     Temp.start_state := value;
                     return;
                  end if;
         end if;
      end loop;

   end Set_Initial_State;
   ------------------------------------------------------------------

   ------------------------------------------------------------------
   procedure Add_State_Variable_To_Equivalency_List(
         State_Variable1 : Wide_String_Ptr;
         State_Variable2 : Wide_String_Ptr) is

      Temp : Signal_Equivalency_List_Ptr;

   begin
      Temp := new Signal_Equivalency_List;
```

```
      Temp.Signal_Name := State_Variable1;
      Temp.equivalent_Signal := State_variable2;
      Temp.next := State_Equivalency_list;
      State_Equivalency_List := Temp;

   end Add_state_variable_To_Equivalency_list;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   procedure Find_Equivalent_States is

      temp_int : integer;
      first_word : wide_String_ptr;
      S_Type : Statement_Type;
      value : wide_string_ptr;
      Temp : State_variable_Ptr;
      Variable_Type1 : Conditional_Type;
      Variable_Type2 : Conditional_Type;

   begin

      Ada.Text_Io.Put_Line("Find_Equivalent_States: ");
      Reset(Input, In_File);
      loop
         First_Word := Next_Word(Input,' ');
         if (First_Word = null) then
            return;
         end if;

         if
(Compare_Words(Wide_String_To_String(First_Word.all,1),"CASE")) then
            find_end_for(Input,"case");

elsif(Compare_Words(Wide_String_To_String(First_Word.All,1),"IF")) then
            temp_int := Find_Next_Matching_Word(Input,"THEN");

elsif(Compare_Words(Wide_String_To_String(First_Word.all,1),"ELSIF"))
then
            temp_int := Find_Next_Matching_Word(Input,"THEN");

elsif(Compare_Words(Wide_String_To_String(First_Word.all,1),"ELSE"))
then
            temp_int := Find_Next_Matching_Word(Input,"THEN");

elsif(Compare_Words(Wide_String_To_String(First_Word.All,1),"END"))
then
            Get_until(Input,';',word,word_length,eof);
         else
            Get_until(Input,';',word,word_length,eof);
            Value := new
Wide_String'(Characters.Handling.To_Wide_String(Word(1..Word_Length)));
            Variable_Type1 :=
Type_Of_Conditional(First_Word,Signal_List,Input_List_Head_Ptr,Output_L
ist_Head_Ptr);
            case Variable_Type1 is
               when State_Variable =>
```

```
                  Variable_Type2 :=
Type_Of_Conditional(Value,Signal_List,Input_List_Head_Ptr,Output_List_H
ead_Ptr);
                  case Variable_Type2 is
                     when State_Variable =>

add_state_variable_to_Equivalency_list(First_Word,Value);
                     when others =>
                        Ada.Text_Io.Put_Line(" ");
                  end case;
               when others =>
                  Ada.Text_Io.Put_Line(" ");
            end case;
         end if;
      end loop;

   end Find_Equivalent_States;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   procedure Print_Equivalent_States is

      Temp : Signal_Equivalency_List_Ptr;


   begin

      Ada.Text_Io.Put_Line("Equivalent_States: ");
      Temp := State_Equivalency_List;
      loop
         exit when Temp = null;
         Ada.Text_Io.Put_Line("EQUIVALENCY >>
"&Wide_String_To_String(Temp.Signal_Name.All,1)&" =
"&Wide_String_To_String(Temp.Equivalent_Signal.All,1));
         temp := temp.next;
      end loop;

   end Print_Equivalent_States;
   ------------------------------------------------------------------


   ------------------------------------------------------------------
   function Find_State_Variable (
         State_variable_list  : State_variable_Ptr;
         State_variable_Name : Wide_String_Ptr )
     return State_variable_Ptr is

      Temp : State_variable_Ptr;

   begin

      Temp := State_variable_list;
      loop
         exit when Temp = null;

if(Compare_Words(Wide_String_To_String(Temp.Signal_Name.All,1),Wide_Str
ing_To_String(State_variable_name.All,1))) then
```

```
            return Temp;
         end if;
         Temp := Temp.Next;
      end loop;

      Ada.Text_Io.Put_Line("ERROR: FIND_STATE_VARIABLE, STATE_VARIABLE
NOT FOUND: "&Wide_String_To_String(state_variable_name.All,1));
      return null;

   end Find_State_Variable;
   ------------------------------------------------------------------------


   ------------------------------------------------------------------
   function Find_Row (
         Row_Head : Row_Ptr;
         State_Name : Wide_String_Ptr )
     return Row_Ptr is

      Temp : Row_Ptr;

   begin

      Temp := Row_Head;
      loop
         exit when Temp = null;

if(Compare_Words(Wide_String_To_String(Temp.Present_STate.All,1),Wide_S
tring_To_String(State_name.All,1))) then
            return Temp;
         end if;
         Temp := Temp.Next;
      end loop;

      return null;

   end Find_Row;
   ------------------------------------------------------------------


 --  procedure Simple is
 --     Window : Gtk.Window.Gtk_Window;
--   begin

---      Gtk.Main.Init;
--       Gtk.Window.Gtk_New(Window);
--       Gtk.Window.Show(Window);
--       Gtk.Main.Main;
--   end simple;



   State           : Wide_String_Ptr;
   Conditional_list : Conditional_Node_Ptr;
   Temp_Int : Integer;
   State_variable_name: wide_string_ptr;
```

```
   type list_of_states;
   type list_of_states_ptr is access list_of_states;
   type List_Of_States is
      record
         State_Variable_Name : Wide_String_Ptr;
         next : list_of_states_ptr;
      end record;
   List_Of_States_Head : List_Of_States_Ptr;
   list_of_states_temp_ptr : list_of_States_ptr;
   Last : Positive;

begin

   ada.Text_IO.Put_line("VHDL TO FSM TRANSLATOR");
   Ada.Text_Io.Put_line(" ");

   ada.Text_IO.Put("PLEASE ENTER FILE NAME TO PROCESS: ");
   Get_Line(Word,Word_Length);
   ada.Text_IO.Put_Line(word(1..word_length));
   Open(Input, In_File, word(1..word_length));

   ada.Text_IO.Put_line(" ");
   ada.Text_IO.Put("DO YOU WANT TO SEE SIGNAL VALUES? (y/n):  ");
   Get_Line(Word,Word_Length);
   if(Word(1..Word_Length) = "y") then
      Signal_Flag := True;
   end if;

   ada.Text_IO.Put_line(" ");
   ada.Text_IO.Put("DO YOU WANT TO SEE OUTPUT VALUES? (y/n):  ");
   Get_Line(Word,Word_Length);
   if(Word(1..Word_Length) = "y") then
      Output_Flag := True;
   end if;

   ada.Text_IO.Put_line(" ");
   ada.Text_IO.Put("Zoom Factor?:  ");
   Get_Line(Word,Word_Length);
   Ada.Float_Text_Io.Get(Word(1..Word_Length),Zoom_Factor,Last);
   Ada.Float_Text_Io.Put(Float(Double(Zoom_Factor)));


Input_Output_Variables(Input,Input_List_Head_Ptr,Output_List_Head_Ptr);
   Signals(Input,Signal_List);

   Print_Signal_Names(Signal_List);
   Print_Output_Names(Output_List_Head_Ptr);
   Print_Input_Names(Input_List_Head_ptr);



   ------------------------------------------------------------------
   -----THE CODE BELOW IS THE ONLY ONE WITH ASSUMPTIONS ---------------
   ------------------------------------------------------------------

   ada.text_io.Put_Line("FINDING EQUIVALENT STATES");
   FInd_EQuivalent_States;
```

```
   Print_Equivalent_States;

   Find_Word(Input,"CASE",Eof);
   if(Eof) then
      Ada.Text_Io.Put_Line("NO STATE MACHINE FOUND!");
   return;
   end if;
   conditional_list := null;

Process_Case_Statement(Input,null,null,State_Variable_Name,Signal_List,
Input_List_Head_Ptr,Output_List_Head_Ptr);
   List_Of_States_Temp_Ptr := new List_Of_States;
   List_Of_States_Temp_Ptr.state_variable_name := State_variable_name;
   List_Of_States_Temp_Ptr.Next := List_Of_States_Head;
   List_Of_States_Head := List_Of_States_Temp_Ptr;
   loop
        Temp_Int := Find_Next_Matching_Word(Input,"CASE");
        exit when Temp_Int = -1;

Process_Case_Statement(Input,null,null,State_Variable_Name,Signal_List,
Input_List_Head_Ptr,Output_List_Head_Ptr);
        List_Of_States_Temp_Ptr := new List_Of_States;
        List_Of_States_Temp_Ptr.state_variable_name :=
State_variable_name;
        List_Of_States_Temp_Ptr.Next := List_Of_States_Head;
        List_Of_States_Head := List_Of_States_Temp_Ptr;
   end loop;


   ada.text_io.Put_Line("FINDING INITIAL STATES");
   list_of_states_temp_ptr := list_of_states_head;
   loop
      exit when list_of_states_temp_ptr = null;
      Set_Initial_State(List_Of_States_Temp_Ptr.State_Variable_Name);
      List_Of_States_Temp_Ptr := List_Of_States_Temp_Ptr.Next;
   end loop;
   ------------------------------------------------------------------------
   ---------- ASSUMPTIONS END ABOVE -------------------------------------
   ------------------------------------------------------------------------


   -- state := new
Wide_String'(Characters.Handling.To_Wide_String("YO"));
   --  Next_State := State;
   -- Put_Line(Wide_String_To_String(state.all,1));
   --  Put_Line(Wide_String_To_String(next_state.all,1));
   -- State := new
Wide_String'(Characters.Handling.To_Wide_String("hey"));
   -- Put_Line(Wide_String_To_String(state.all,1));
   -- Put_Line(Wide_String_To_String(next_state.all,1));

   -- ppp := new Wide_String'(Characters.Handling.To_Wide_String("YO"));
   --  qqq := ppp;
   -- Put_Line(Wide_String_To_String(ppp.all,1));
   --   Put_Line(Wide_String_To_String(qqq.all,1));
   -- ppp := new
Wide_String'(Characters.Handling.To_Wide_String("hey"));
```

```
   --   Put_Line(Wide_String_To_String(ppp.all,1));
   --   Put_Line(Wide_String_To_String(qqq.all,1));


      Print_Table_Info(State_Variable_list);
      Create(Output, Out_File, Output_File_Name);
      Fill_In_File;       -- uses data structures to create XML file


    Close(Output);
end parser;
-------------------------------------------------------------------------
-------------------------------------------------
```