A METHOD FOR MODELING AND VERIFICATION OF REAL-TIME SYSTEMS

By

Jason Matthew Scott

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

December 1997

Nashville, Tennessee

Approved:	Date:

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Gabor Karsai for his support and guidance in this research over the past year and a half. Dr. Janos Sztipanovits, Dr. Csaba Biegl, and Dr. Karsai are responsible for the MultiGraph Architecture from which this work stems and made this work possible.

I would also like to thank Dr. Ted Bapty and Richard (Bubba) Davis for all of their help and support. Other members of the Measurement and Computing Systems Laboratory have also been supportive: Dr. Akos Ledeczi, Dr. Amit Misra, and Michael Moore. Thanks also to Captain Greg Nordstrom for selecting me for a summer internship seven years ago which resulted in my involvement with this group.

Thanks to my family which has always provided much encouragement. Thanks most of all to my wife, Amy. She has given me a great deal of support throughout the last four years. Without her support this would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Some of the most important applications of computers are embedded real-time systems. These systems are used in such areas as avionics, medical equipment, plant control, etc. Most software systems attempt to provide correct results in an efficient manner. Real-time software systems not only must be numerically correct, but also have additional specifications for the time in which the results must be delivered. Correct results are useless if, by the time they are delivered, they are too late to provide the feedback necessary to prevent system damage or failure. Human lives or valuable equipment may be at risk in the event of a failure. Real-time systems have well-defined, rigid timing constraints; they must process events within a given time frame [16]. It is critical that real-time systems be correct in both time and value. A real-time system is considered to function correctly only if it returns the correct results within the associated time constraints. Processing *must* be done within the defined constraints or the system will fail.

Problem of Real-Time System Verification

Real-time systems are classified as hard real-time systems or soft real-time systems. A *hard* real-time system must meet its time critical deadlines or catastrophic system failure may occur. A less restrictive type of system is a *soft* real-time system. Soft real-time systems should meet their deadlines but a catastrophic failure will not occur if the deadline is missed. An example of a soft real-time system is a video conferencing system. The system has a fixed time frame in which the images must be displayed, but if an images is displayed late it may not be noticeable to the person viewing the system.

Hard real-time systems must meet their deadlines to provide correct operation. Processing must be completed on time with the correct results. If these processing constraints are not satisfied in critical applications the results could be disastrous. In systems such as data acquisition systems for jet engine testing, failure to meet the deadlines could mean loss of very expensive test data.

It is clear that a method to verify the correctness of these hard real-time systems is necessary. As the complexity of the system increases, the number of possible ways the system can execute grows exponentially in size and can quickly become too large to test. Presented here is a way to model real-time systems for the purpose of verification. A method is presented to efficiently explore the very large state spaces that occur.

Simulation techniques are often used to verify real-time systems. These simulations do not always expose problem areas because all possible system states may not be explored. Theorem provers and proof checkers are also used to verify systems although these methods are very time consuming and labor intensive [4]. These methods can become very complex for large systems. One approach to addressing how to determine whether a group of processes, whose individual CPU utilization is known, will meet their deadlines is real-time scheduling theory.

Real-Time Scheduling Theory

Real-time scheduling theory addresses the issues of priority based scheduling of processes with hard deadlines. Priority driven scheduling algorithms have been widely used in single processor and multi-processor systems. In this method, each process is assigned a priority. This priority serves to give preference for execution over other processes with lesser priority. The highest priority process that is ready to run is the next process selected for execution. In a *static* priority algorithm the priority is assigned to the task only once. A *dynamic* priority scheme may change the assignment of priorities with time.

Rate monotonic scheduling (RMS) [5] is a static algorithm that assigns priorities to a set of independent, periodic real-time tasks based on their periods. Each process is characterized by a period, T_i and an execution time C_i . Tasks are executed repeatedly and each invocation of a task must complete before the beginning of the next period. The algorithm assigns higher priority to tasks with a shorter period. The RMS algorithm has been proven to be optimal among all fixed-priority algorithms. If there exists a fixed priority assignment that can generate a feasible schedule, then the RMS algorithm can produce a feasible priority assignment. If processor utilization remains below a certain level then this priority assignment will assure that the processes will meet their deadlines. [15]

This scheduling algorithm guarantees only *average* performance [17]. The system can fail to meet deadlines during transient peak overloads where the processor utilization bounds are exceeded. Also, a non-critical task with a high frequency may postpone a critical task with a low frequency. This case is known as *priority inversion*. This phenomenon is possible even if the processor utilization remains below a maximum bound if a resource conflict occurs. This occurance is a serious problem because it causes the behavior of the real-time system to seem unpredictable. In real-world problems, situations occur where the rate monotonic assumptions do not hold. Although the RMS algorithm produces optimal results, the use of scheduling algorithms alone cannot guarantee that a system will always meet its deadlines.

Computational Tree Logic

S. Campos describes a formal method for modeling real-time systems [4]. He uses Symbolic Model Checking as a method to verify these real-time systems. Symbolic Model Checking is the problem of determining whether a given logic formula f is true for a supplied state transition graph [6]. This state transition graph is a model of the behavior of the system. The formula is tested to be true using graph traversal techniques on this state graph.

This symbolic model checking checking approach was extended to verify the properties of real-time systems. The model checker receives a specification of the system in the form of a propositional temporal logic. The system to be verified is modeled as a state transition graph. The model checking process traverses the state-transition graph, testing to see if it satisfies the given properties.

The properties of the system are represented as formulas in a temporal logic known as Computational Tree Logic (CTL). Formulas are built up using CTL to describe properties of the system that is being modeled. The formulas consist of

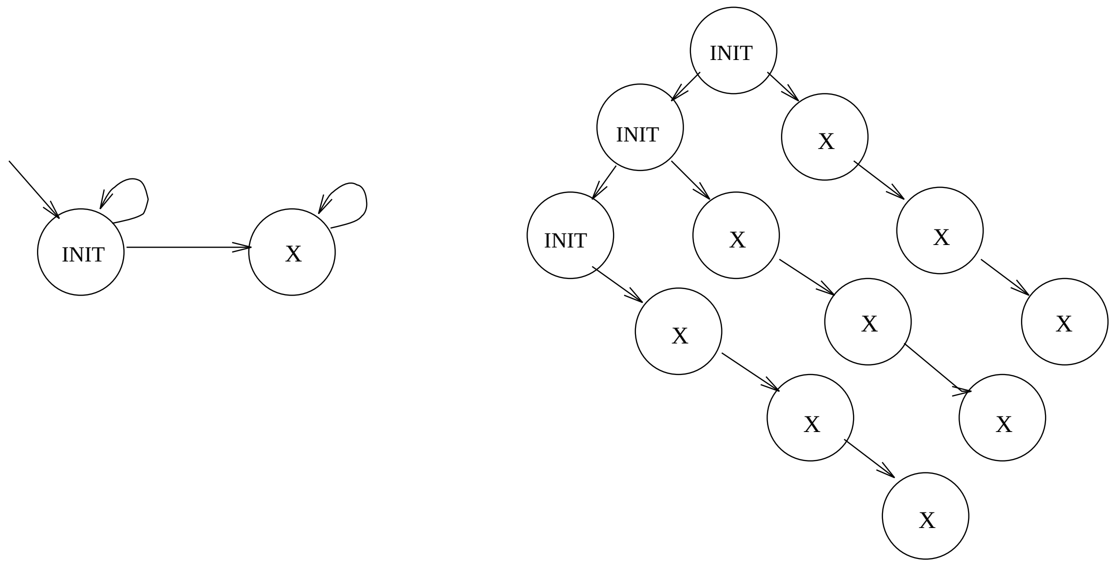


Figure 1: State graph and corresponding computation tree

atomic propositions, boolean connectives, and temporal operators.

The CTL logic can state properties such as "event p will happen sometime in the future". For use in specifying real-time systems the CTL was augmented with operators which represent deadlines such as "event p will happen in at most t units of time".

Temporal logic model checking is a method for verifying the correctness of finite state transition systems. Specifications of the system to be verified are written as formulas in temporal logic. The model checker traverses the state transition graph to verify if the model satisfies the given properties. Because checking the model to satisfy a given formula is much simpler than proving that a formula is valid for all possible models, the model checker is much more efficient than a theorem prover [3].

Computation trees are derived from state transition graph models of the system. The state transition graph is expanded into an infinite tree structure. The states are unrolled such that no states are revisited. The paths in this tree represent all execution possibilities in the system that is being modeled. Figure 1 shows the relationship between a state diagram and its corresponding computation tree. The root of the computation tree is the initial state in the transition graph. The CTL formulas refer to the computation tree that is derived from the system model.

The complexity of this algorithm is linear in the size of the unwound state transition graph and the length of the formula to be proven. For some systems, however, a very large state-transition graph may be needed to model the system. When this type of state explosion occurs the representation may become too large to manage. To help solve this problem Ordered Binary Decision Diagrams (OBDDs) are used for the internal representation of the state-transition graph. Using this symbolic method of representation avoids the enumeration through all states of the graph. This implementation, as we will see in the next chapter, provides a very efficient method for manipulation of these expressions.

The Approach

A model-based approach is selected for verifying real-time systems. Using a model-based approach allows the system to be designed and analyzed before it is actually constructed. This can allow the designer to perform "what if" situations and analyze different designs without actually implementing them.

Model-based software design has been successfully used to reduce the complexity in large software systems. Model-based analysis tools can use these same models. Additional information is added to the models to give the timing characteristics of the system.

The system model is a dataflow graph. The processes are represented as data-driven process blocks in the dataflow graph. The model of the system is translated into a Finite State Machine (FSM) representation. Ordered Binary Decision Diagrams (OBDDs) are used to represent the FSM. The OBDDs provide an efficient way of traversing the large state space that results from this representation of the system [4]. Algorithms are developed to verify that the system *always* meets its timing constraints to provide correct operation.

The next chapter describes a successful model-based programming system. Ordered Binary Decision Diagrams, Finite State Machines and a modified version of the FSM are defined also. Chapter III explains the technique for the verification of real-time systems developed in this research. An algorithm for constructing a FSM representation of a real-time system and algorithms for searching for the longest and shortest paths between two states on a FSM are described. Chapter IV presents the verification and timing analysis of a practical system. Chapter V concludes the paper.

CHAPTER II

BACKGROUNDS

Model-Based Programming

Just as models are a fundamental part of other areas of science and engineering, model-based techniques can be useful in software engineering. Model-based programming can facilitate management of complex software systems and enable easy system modification and generation. Applications are generated from models which specify the system. Model-based analysis can be performed while the system is still in the design stages to gain knowledge of the system's timing characteristics. The MultiGraph Architecture (MGA) supports a model-based method for the generation and analysis of software and systems. The MultiGraph Architecture was developed at Vanderbilt University and has had success in several application areas [1].

The basic concepts behind the design of MGA involve the use of domain-specific system modeling. By modeling a system in terms familiar to the user, some of the complexity of the system can be hidden from the user. The software produced will actually follow model of the system it is generated from unlike other design methods in which too much freedom is given between the design and implementation stages. Model-based analysis can be performed while the system is still in the design stages to gain knowledge of the system's timing and other characteristics. A diagram of the overall approach provided by MGA is shown in Figure 2.

The specifications and constraints of the system to be produced are specified by

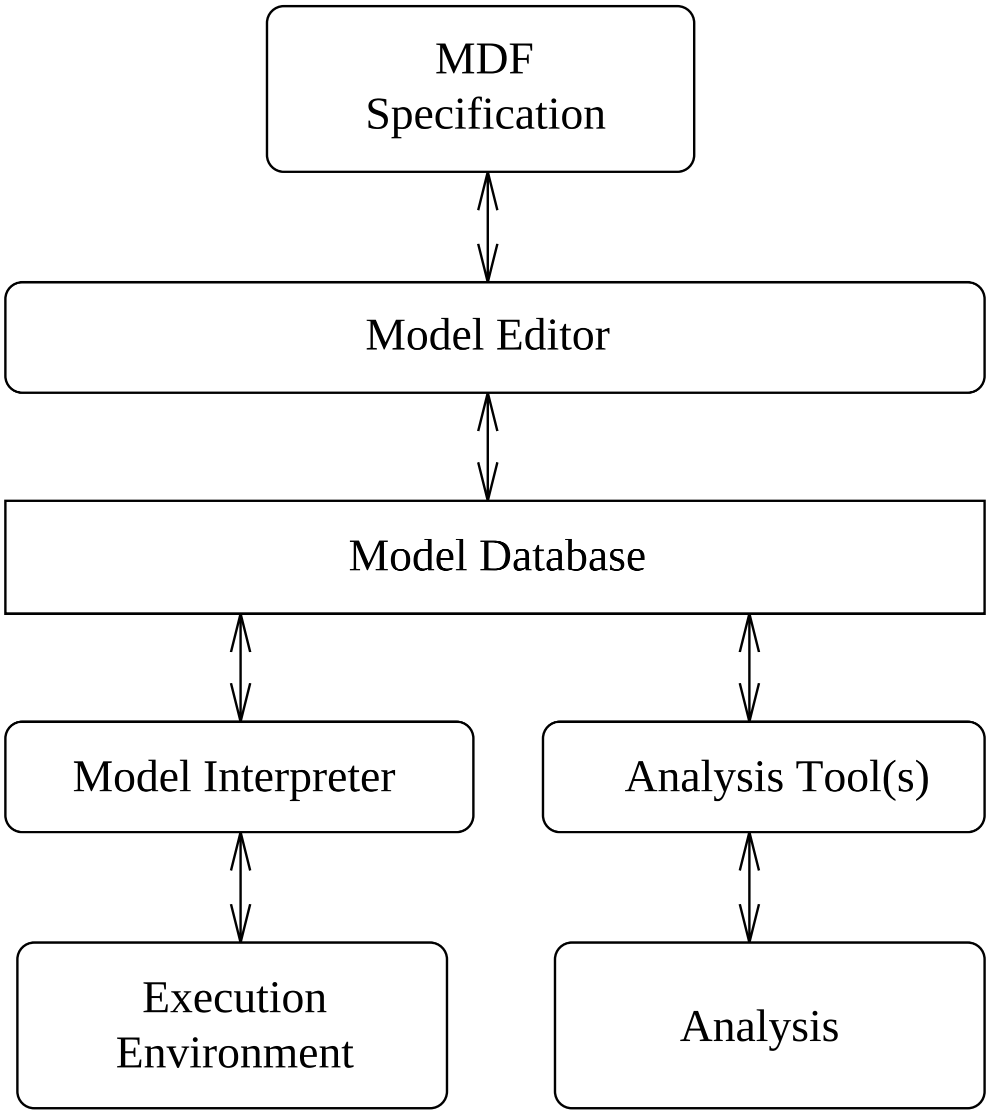


Figure 2: Overview of MGA System Structure

a Model Definition File (MDF). This file is written in a declarative language for the purpose of specifying model domain concepts such as: application specific objects to be used in the model, connections that are allowed between objects, non-graphical parameters that are also used to describe the objects, etc. The MDF file used in this application and its description is given in appendix A. Creating a MDF file the the first step in building a new system. Using the *builder* program, a Model Editor executable is automatically custom built from the specification described by the MDF file.

Model Editor

The XVPE Model Editor provides a graphical means for specifying the system model as shown in Figure 3. The Model Editor is automatically generated for each specific modeling paradigm. The objects available for use and their attributes are specified in the MDF file. Valid connections that may be made between objects are also specified in the MDF file.

Model Editor supports different model *aspects*. The aspects are useful for representing different parts of a system. A different set of modeling objects may be used for each aspect. For example, one aspect could represent a signal flow model for a signal processing application and another aspect of the same system could model the physical hardware it is to be implemented on. *References* are provided to model interactions between different aspects. The models may be also hierarchical: A model may be composed of other models.

The entire model entered into the Model Editor is stored in an object oriented

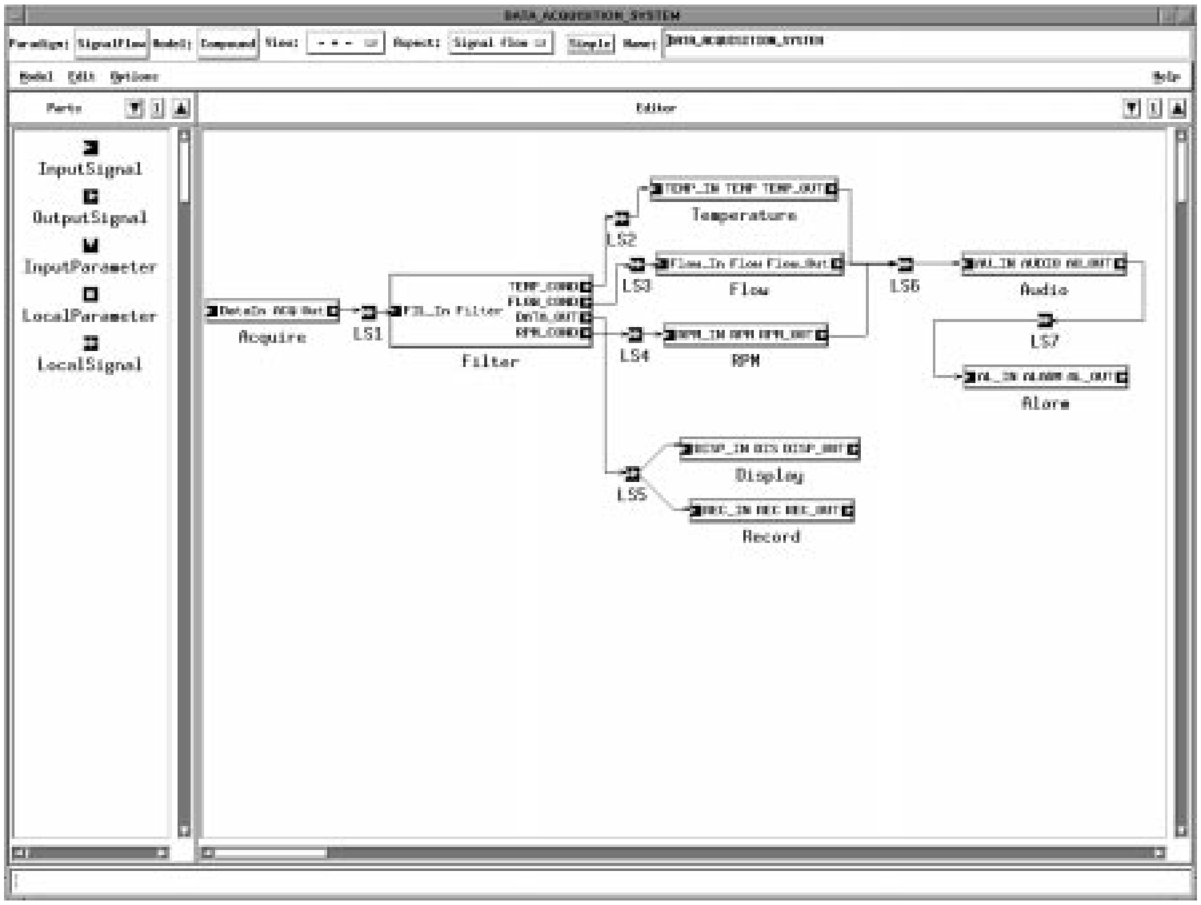


Figure 3: XVPE Model Editor

database. Exact implementations of this database vary depending on which platform the MGA system is implemented.

MGA Interpreter

The purpose of the interpreter is to extract the models from the database for some useful purpose, usually to build a runtime system or analysis system. The interpreter synthesizes executable program structure from models retrieved from the database by building the dataflow graph and setting the parameters of the individual process blocks. The model interpreter is specific to the modeling paradigm used.

The interpreter also generates data structures and other output used for tools that perform various types of analysis on the systems that are to be built.

MGA Dataflow representation

A *dataflow network* consists of process blocks and connections. A process block is not available for execution unless it has data available on its inputs. When a process block is executed its output data flows along the connections to place data on the inputs of another process block and cause it to execute. This graph is used to represent a data-driven system. Data input to the system triggers the execution of the process(s) blocks that the inputs are connected to. These process blocks, in turn, cause the execution of other process blocks.

The *MultiGraph Computational Model* representation modifies the standard dataflow graph. The MGA dataflow graph contains *actor nodes*, *data nodes* and connections between these elements. Actor nodes are the processing elements of the system. The actor nodes are connected with a data node between them. A data node serves as a buffer for the data. This buffer is of a specified length (specified in the Model Editor).

Each process block (actor node) contains:

- a script — a reentrant algorithm that performs some processing on its inputs.
- I/O Ports — the I/O data streams
- Control discipline — determines under what input conditions an actor should be executed. An IF_ALL discipline will mark an actor ready to execute if data is available on *all* of its input ports. IF_ANY means the actor should be ready

to execute if data is present on *any* of its input ports.

The dataflow graph structure defines data dependencies between the actor nodes. The execution order is determined at runtime based on these dependencies and the actor nodes control disciplines.

MGA Kernel

The Multigraph Kernel (MGK) provides the execution layer for MGA. The kernel uses the output of the model interpreter to build a MGA dataflow network composed of actor nodes and data nodes. The MGK implements a data-driven system that is derived directly from a dataflow model.

The kernel provides a layer of abstraction for a network built using heterogeneous hardware (parallel computers, networked single processors, embedded processors, etc.)

The MGK has also been re-implemented as a micro-kernel. This Modular Kernel [7] is smaller is size and more application specific. It provides basic kernel components: real-time scheduler, communications between adjacent processors, and memory management. These key components are tightly coupled for efficient operation. This kernel can be easily ported to new architectures.

MGA Application Domains

MGA has been successful in many application areas. These applications represent a variety of engineering disciplines from turbine engine testing to plant control.

- The Computer Assisted Dynamic Data Measurement and Analysis System (CADDMAS), a large parallel processing instrumentation system has been developed for high speed turbine engine testing at Arnold Engineering and Development Center [11].

- Intelligent Process Control System (IPCS), a chemical engineering monitoring and control system [1].

- Model-Integrated Real-Time Imaging System (MIRTIS), a high-speed image processing system implemented with a network of parallel processors [12].

Systems considered

The real-time system is assumed to be data driven and is modeled by a dataflow diagram. Specifically, the MGA dataflow graph model is used to represent the system. Real-time systems that are candidates for verification by this tool are assumed to have one periodic event such as a process which acquires a new data sample. All other processes scheduled to run during this period between which the data samples are taken must finish before the next sample period begins. The system is schedulable if and only if all processes are guaranteed to finish execution before the next sample period begins.

In this data-driven system, a process is scheduled for execution only if data is available at its inputs and the specified control discipline (IF_ALL, IF_ANY) is satisfied. Processes are chosen for execution from this schedule by their assigned priority. The highest priority process is selected. The systems considered are those with N

number of processes each with a statically assigned priority and execution time.

This type of data-driven system model is useful for representing many types of systems including signal processing applications, monitoring, etc. and is directly applicable to systems using the MultiGraph Computational Model.

Finite State Machines

Finite State Machines (FSMs) are used to represent dynamic systems where at each moment the system is considered to be in one of a finite number of unique *states*. Since a system can only be in one state at a time, a state represents what is currently happening in the system.

The FSM model has discrete inputs and outputs. The state of the system summarizes the information concerning past inputs that is needed to determine the behavior of the system on later inputs [18]. When a state change occurs the next state is chosen based on the system's inputs and available transitions.

A state machine can be modeled using Boolean algebra. Each state is assigned a unique encoding. A state S is represented by the encoding assigning values to the state variables $s_1, s_2, ..., s_n$. The next state equation $N(s, s')$ evaluates to true when their is a transition in the FSM from the state S to state S', where s is in terms of $(s_1, ..., s_n)$, and s' is in terms of $(s'_1, ..., s'_n)$.

The state machine representation used in this application has nondeterministic transitions. These transitions are simply showing that, under some undefined circumstances, this transition may occur.

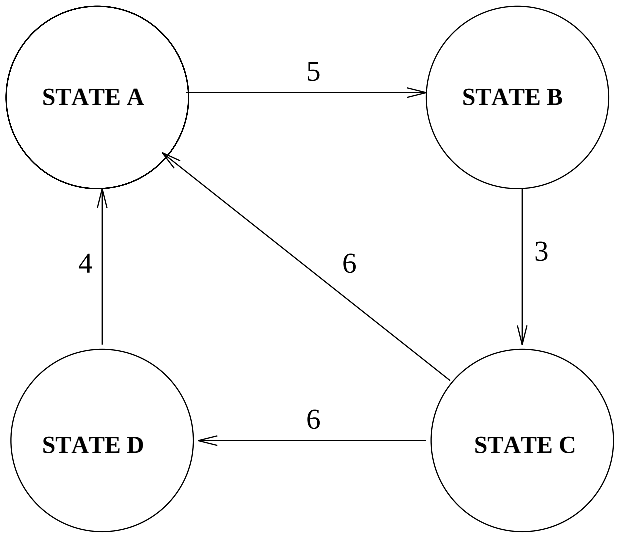


Figure 4: Timed Transition Graph

Timed Transition State Machines

Standard state machines have a big limitation for modeling real-time systems — all transitions happen in one unit-length step. If state machines are to be used to model a real-time system which has events that take various lengths of time to occur then we would like to incorporate this in a modified state machine type representation. One way around this would be to use multiple states in series to represent a transition longer than 1 unit. For example, to represent a 3 unit length transition we could put 3 states in series with only 1 transition between each of these states. For small systems this approach works well, but for large transition lengths and/or many non-unit transitions state explosions problems can occur.

A more efficient approach would be to create a new set of execution rules for a modified state machine. A *Timed Transition State Machine* will have each transition labeled with a time. This time will represent the time it takes for the transition to the next state to occur.

In this application the transition time will represent the time that is spent in a state as shown in Figure 4. Note that all of the transitions leaving a state should have the same time since this time actually represents the execution time of the state.

This approach does add considerably to the complexity of the execution of the state machine, but its compact representation compensates for this. Also this compact representation helps to reduce state explosion problems.

Ordered Binary Decision Diagrams

Ordered Binary Decision Diagrams (OBDDs) [2] are a canonical representation for Boolean functions. They represent the function in the form of a rooted, directed acyclic graph. Mathematical operations on Boolean formulas can be implemented as graph algorithms operating on OBDDs. OBDDs work well for representing truth tables, search trees, and state transition graphs (as we will see). The advantage of using OBDDs is that the entire state space of a problem never needs to be constructed. This has allowed problems to be solved that would not be possible to solve using exhaustive search. OBDDs are used in the application presented in this paper to represent and analyze finite state machines which may be very large. OBDDs have already had success in many types of verification and digital design [2].

The vertices of the graph are organized in levels by index. The internal vertices of a graph on the same level correspond to a particular variable of the Boolean function. The edges of the graph are label with 0 or 1. For each variable assignment for a corresponding path in a OBDD at vertex x the edge labeled 1 is taken is the variable

x is to be set to 1; if x is to be set to 0 the 0 edge is taken. All paths of the graph lead to one of the two terminal vertices labeled by the values 0 and 1. If a path taken from the root node ends at the 0 terminal node then the assignment does not satisfy the equation represented by the OBDD. If the 1 terminal node is reached then the equation is satisfied by the assignment. The graph forms a canonical representation of the function — for a given variable ordering, two OBDDs for a given function are isomorphic. This means equivalence can easily be tested. Figure 5 illustrates the OBDD for the Boolean formula $(a \vee b) \wedge \overline{c}$ using the variable ordering $a < b < c$.

The ordering of the variables (the ordering of the levels within the graph) can have a dramatic effect on the size of the OBDD. For each possible variable for a given Boolean expression there exists a unique OBDD [2]. A bad variable ordering can greatly increase the size of the OBDD and, in turn, increase the time required to perform operations on that OBDD. Sometimes, more importantly, the OBDD may be too large to fit in the computer's physical memory. For some choices of variable ordering the representation of a function can grow exponentially as the number of variables grow, while for other choices of ordering the size may only be of linear growth. Finding an optimal variable ordering is a NP-complete problem. However, by using common sense in choosing the variable ordering these problems can usually be avoided and a OBDD that grows linearly may be found.

OBDD Algorithms

Many efficient algorithms for manipulating Boolean functions are available. There are BDD algorithms for all the basic Boolean operations such as AND, OR, NOR,

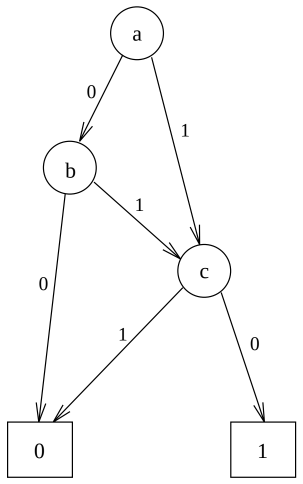


Figure 5: OBDD for $(a \vee b) \wedge \overline{c}$

NOT, and XNOR. Operations performed on two OBDDs, f and g produce a resultant function h (in the form of an OBDD). The *restrict* algorithm is used to simply assign a constant value c to a variable x in function f:

$$f|_{x \leftarrow c}$$

This result of this algorithm is a new function that no longer has the variable x, but has been replaced by the constant c and simplified.

Several other algorithms rely on the restrict function. The *composition* algorithm is used to substitute a function (or variable) g for variable x of function f.

$$f|_{x \leftarrow g} = \overline{g} \cdot f|_{x \leftarrow 0} + g \cdot f|_{x \leftarrow 1}$$

Another operation is the *existential quantification* algorithm:

$$\exists x f = f|_{x \leftarrow 0} + f|_{x \leftarrow 1}$$

This algorithm is used where the variable x in the function f is to be set to “don’t care” or essentially removed from the equation.

Using OBDDs to Represent State Machines

As seen in the preceding section, a finite state machine can be described by its next-state equation. For a FSM whose state is determined by present-state Boolean variables, $S = s_0, s_1, ..., s_{n-1}$ and the next state is described by $S' = s'_0, s'_1, ...s'_{n-1}$, then $N(s_0, .., s_{n-1}, s'_0, ..., s'_{n-1})$, where n = $\lceil log_2($ max number of states $)\rceil$ describes the relationship between the current state and next state. These Boolean variables $(s_0, s_1, .., s_{n-1})$, and (s'_0, s'_1, s'_{n-1}) actually represent a sets of states. Given a state set S and a next state equation N then a next state set S' can be found by computing the intersection of S and N. The intersection of S and N actually gives the OBDD of $S \vee S'$. What we really want is a function that will return the next state (set) of S in terms of the variables $(s_0, s_1, .., s_{n-1})$. A function that will implement this is shown below:

```
FindNextState(S)
{
    $\text{Temp}_1(s_0, ..., s_{n-1}, s'_0, ..., s'_{n-1}) = \text{S}(s_0, ..., s_{n-1}) \cap \text{N}(s_0, ..., s_{n-1}, s'_0, ..., s'_{n-1})$
    $\text{Temp}_2 = \text{Temp}_1|_{s_0 \leftarrow 0} + \text{Temp}_1|_{s_0 \leftarrow 1}$
            $+ \text{Temp}_1|_{s_1 \leftarrow 0} + \text{Temp}_1|_{s_1 \leftarrow 1}$
    $\text{Result} = \overline{s_0} \cdot \text{Temp}_2|_{s'_0 \leftarrow 0} + s_0 \cdot \text{Temp}_2|_{s'_0 \leftarrow 1}$
            $+ \overline{s_1} \cdot \text{Temp}_2|_{s'_1 \leftarrow 0} + s_1 \cdot \text{Temp}_2|_{s'_1 \leftarrow 1}$
    return Result;
}
```

First, the intersection of S and N is computed with the result stored in a OBDD to provide temporary storage, Temp$_1$. This intersection is the union of S and S'. Next, because we are interested in getting S' into an equation alone, the variables s_0 and s_1 are both existentially quantified to remove them from the OBDD. This new equation is stored in $Temp_2$. To return the next state in terms of the present state variables, s_0 and s_1, we must use the composition function to substitute the variable s_0 for s'_0 and s_1 for s'_1. Now our final equation, *Result*, gives us a state set in terms of present state variables. In Figure 6 the intermediate OBDDs of an example equation are shown.

The *and* operation is performed efficiently on the two OBDDs. Similarly, if a state set S' is known, along with the next-state equation, then a previous state set S can be found. This function is similar to the FindNextState() function:

```
FindPrevState(S)
{
    Result = $\overline{s'_0}\cdot\text{Temp}_1|_{s_0\leftarrow 0} + s'_0\cdot\text{Temp}_1|_{s_0\leftarrow 1}$
             $+\overline{s'_1}\cdot\text{Temp}_1|_{s_1\leftarrow 0} + s'_1\cdot\text{Temp}_1|_{s_1\leftarrow 1}$
    $\text{Temp}_2(s_0, ..., s_{n-1}, s'_0, ..., s'_{n-1}) = \text{Temp}_1(s_0, ..., s_{n-1}) \cap \text{N}(s_0, ..., s_{n-1}, s'_0, ..., s'_{n-1})$
    Result = $\text{Temp}_2|_{s'_0\leftarrow 0} + \text{Temp}_2|_{s'_0\leftarrow 1}$
             $+ \text{Temp}_2|_{s'_1\leftarrow 0} + \text{Temp}_2|_{s'_1\leftarrow 1}$
    return Result;
}
```

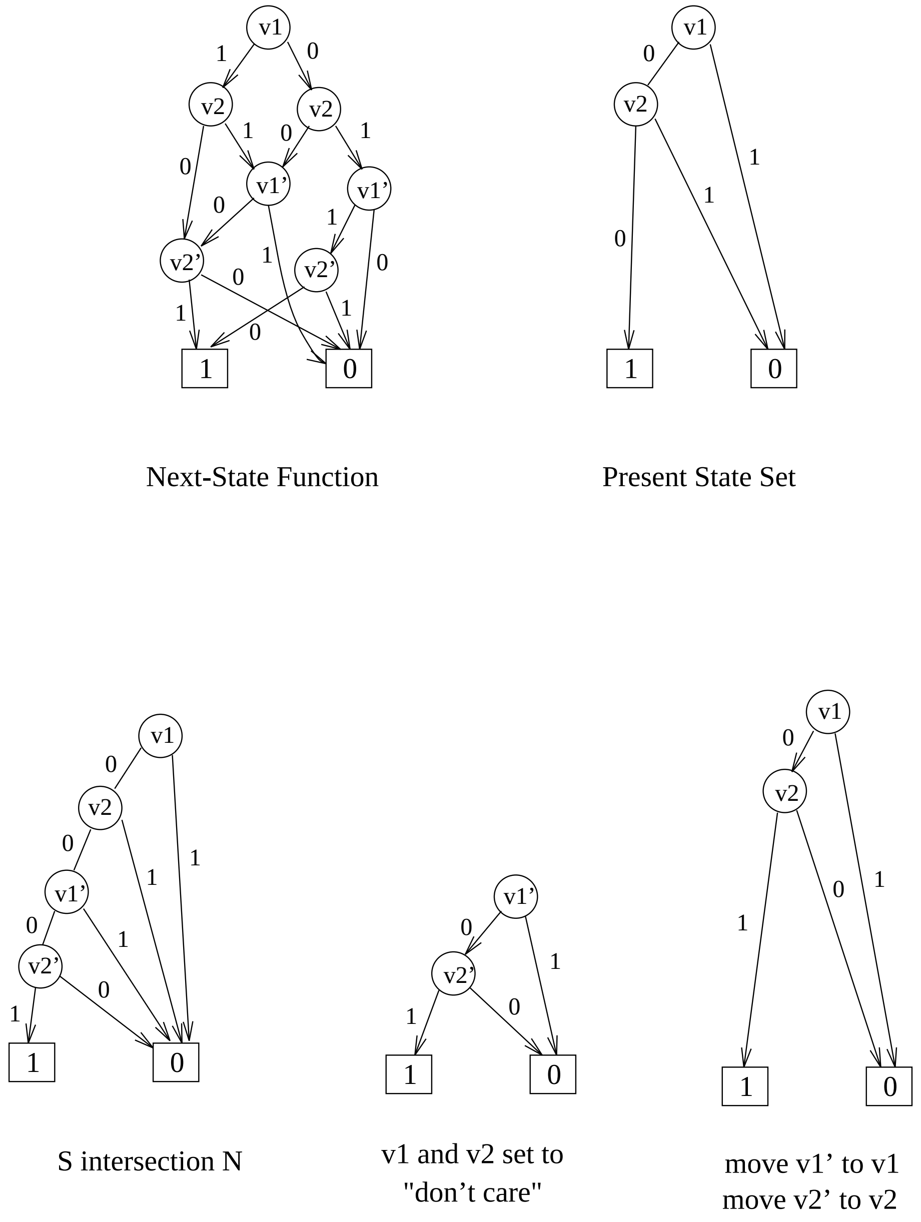


Figure 6: Intermediate OBDDs used in computing next state

This function begins by moving the next state input, S, which is in terms of the variables s_0 and s_1, to be in terms of the variables s'_0 and s'_1. This is done by using the composition function to substitute s'_0 for s_0 and s'_1 for s_1. Next the function just obtained is anded with the next state functions, N. The result is an OBDD that contains the input to the algorithm, S in terms of the variables s'_0 and s'_1 and its corresponding previous state in terms of the variables s_0 and s_1. Lastly, since we are only interested in returning the previous state alone, the variables s'_0 and s'_1 are both existentially quantified to remove them from the OBDD. The result is the previous state of the input. Performing operations on sets of states instead of on individual states provides an efficient method to search a large state space.

It is interesting to note that by using a OBDD to represent this characteristic function we are actually storing relationships between a pairs of states. For a given state, or set of states, we can very efficiently find the set of their corresponding states. This technique could be used to store large amounts of associations between objects.

The power of this technique is that to find the next states of a state or state set a full enumeration of the problem space (eq. the truth table, state graph, etc.) never needs to be constructed [2].

Using OBDDs to Represent Timed State Machines

Timed Transition State Machines are represented by a Boolean equation (and hence, a OBDD) in a similar method as shown above. A time of execution of each state is tacked on the the end of each state encoding. The original state encoding used n bits. The new state encoding will use $n + n_t$ bits, where $n_t = \lceil log_2($ max transition time)] as shown in Figure 7. The top figure is the structure of the bit encoding for a state machine next state equation with unit length transitions. No timing information is stored here. The first n bits on the left side make up the state vector that describes our present state. The last n bits on the right side make up the state vector that describes the next state. This entire 2n bit encoding provides a way for us to store sets of pairs of states where each pair represents a transition from the present state to the next state.

The bottom figure describes the structure of the bit encoding for a timed transition state machine. For each state vector there is an associated transition time. Actually, our next-state equation does not have any knowledge of this added timing information. It only sees a longer state encoding vector. The exact same procedures for using the next state equation to move from one state to a next or previous state are used here also. The only difference is that when we need to know the execution time associated with a state we can look at the entire present state encoding vector and read the bits used for representing the time. These bits are the time stated in binary form. This provides a convenient method of extracting time information from the new, modified state encoding. To find the transition time of a particular state the last n_t bits must be masked off.

The next state equation must be adjusted accordingly to use the new state encodings, but all transitions will remain the same. It may seem strange that the next state time is included in each state transition vector. We do not need to know the time that it takes for the next state to execute. It is included because these times are simple encoded as part of the state encoding that uniquely identifies the state. This

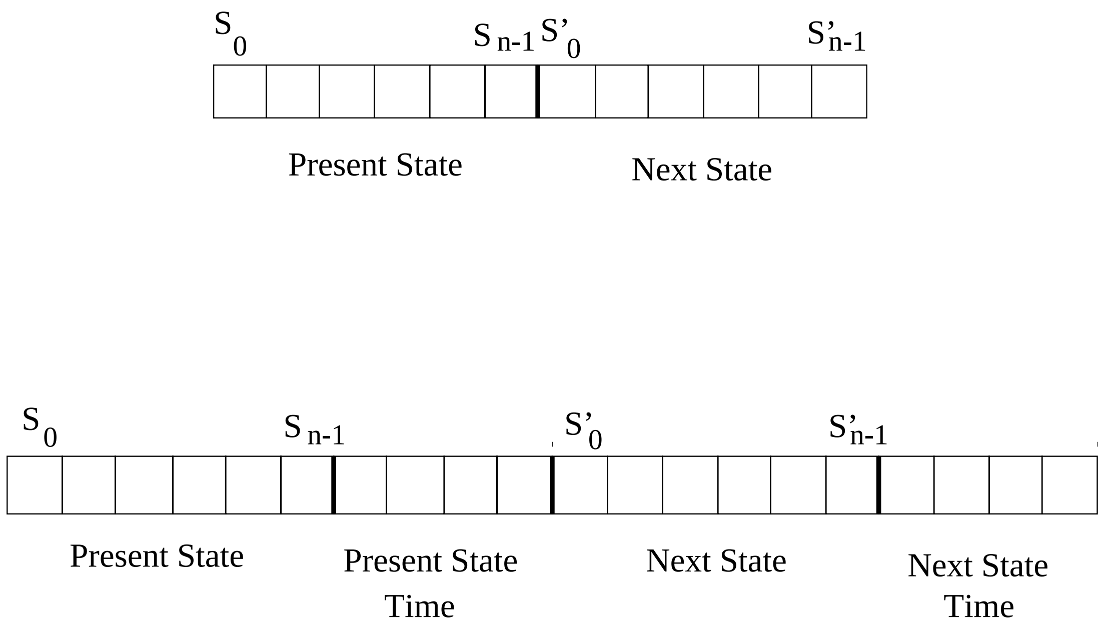


Figure 7: State encoding vectors

new state machine representation operates just as before, storing transitions between pairs of states, only now the state vectors have be lengthened to include an encoding of the states execution time into the state vector itself.

CHAPTER III

MODEL ANALYSIS

Model Representation

The model used to represent the Real-Time system is consistent with the Multigraph Computational Model's execution semantics. The real-time system is modeled as a collection of processes connected in a data driven fashion. One process is marked as the periodic process in the system with the period it executes at. Each process has its own attributes:

- Execution Time — Execution time of process
- Priority — Relative to the other processes
- Data Inputs — Inputs to process
- Data Outputs — Outputs from process (May be conditional or unconditional)
- IF_ANY / IF_ALL —- Processes execution condition

The model process's information may also contain more information for the actual implementation of the real system such as the name of the code, or *script*, that the process will actually run.

In this type of system the execution of a process happens when data is available on its inputs. A process may be triggered by data on its inputs in two different ways as described previously. If it is a IF_ANY process then the process is ready for execution if data is present on *any* of its input ports. If a process is specified as IF_ALL then the process is ready to execute only if data is present on *all* of its input ports. Different types of algorithms may need to be triggered differently. A two input adder block, for example, would need data on both inputs before its execution would make any sense. An alarm process with several inputs may operate by watching each of its inputs for data that is greater than some threshold value. It will test each input as data is available. If the alarm process has to wait on all inputs to become available, and one input does not receive data very often or stops receiving data at all then the others will not be checked frequently enough.

Modeling Procedure

The modeling procedure begins by entering the model into the XVPE Model Editor. The dataflow representation is entered using basic building blocks known as *primitives* to represent processes. Input and Output ports may be added to the primitives as needed. Other attributes of the primitives must also be specified. Each primitive must have a name which describes the type of process it represents or procedure it will execute. The execution time of the process must also be specified in the primitive. As previously described, each process may be marked as an *IF_ANY* or *IF_ALL* type of process. This determines how the process is triggered for execution.

Outputs of a process may be specified to be: *Unconditional* or *Conditional.* An Unconditional output port is used to represent an output of a process that *always* outputs data on this port every time it executed. The Conditional output port may or may not output data every time the process runs based on the data or other parameters not specified in the model. For example, a process may or may not output an alarm condition based on the magnitude of its input data. Both cases (data is output or data is not output) must be taken into consideration when analyzing the system. One node in the system must be identified as the periodic node and its period must be specified.

The system models may be hierarchical. *Compound* objects are higher level modeling objects that may be composed of either primitives, other compounds, or both. The models entered into the editor are stored in a model database. After the model is stored into the database, the model interpreter program may be run to load the model from the database to perform some useful operation with this model information. Interpreters are written specifically to interface with the modeling database they are to be used with. Each model database is specific to the *modeling paradigm* used. The modeling paradigm the definition of model components and their allowed interaction as defined by the MDF. In this application the interpreter's job is to extract the dataflow graph and information associated about each processing element from the database.

The interpreter loads the model from the database and constructs its own internal structure of the model. This internal structure is a network of objects, with each object corresponding to a object in the database. With the model entirely loaded from the database the interpreter can perform operations on this structure.

The model interpreter used in this application constructs a dataflow graph from the model. If the model is a hierarchical structure then the interpreter flattens this

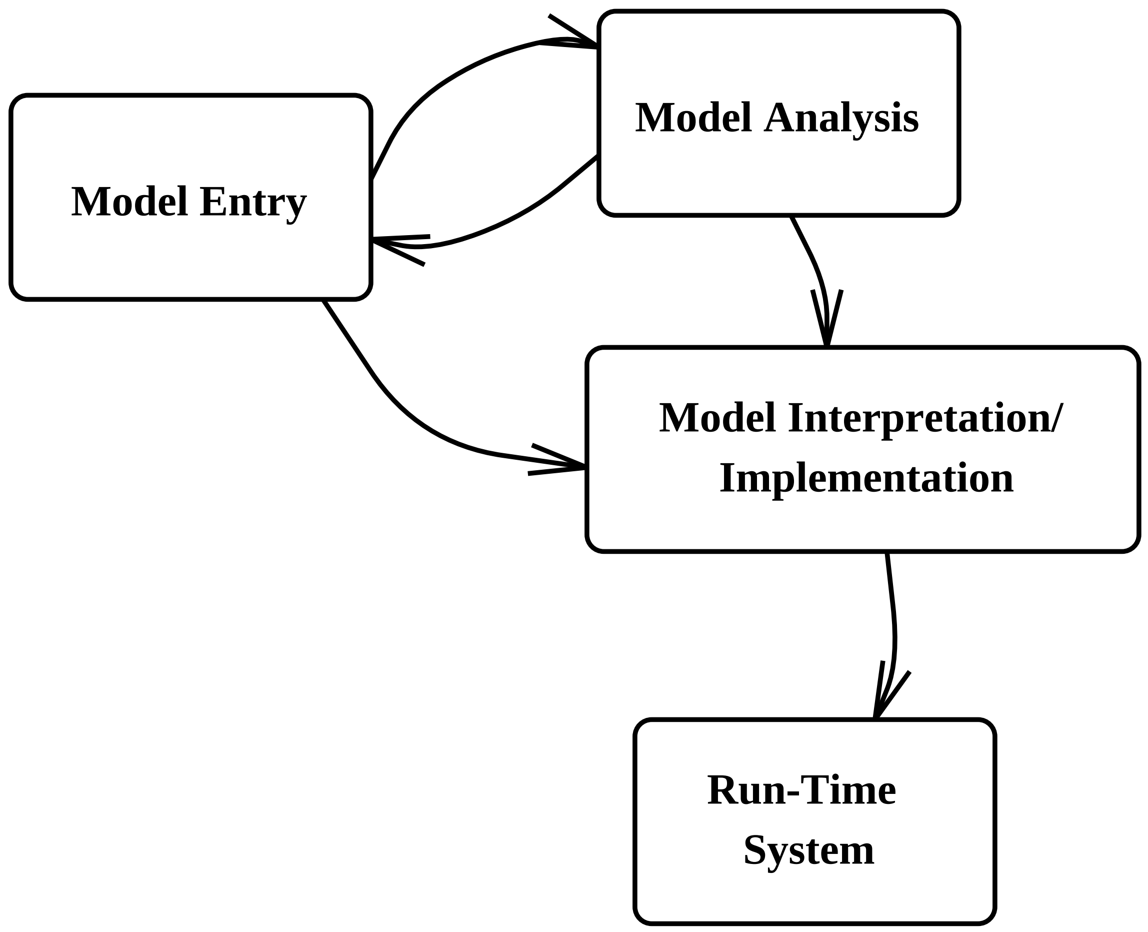


Figure 8: Example of MGA Design Flow

structure to produce a dataflow graph consisting of only processing elements (primitives) and dataflow connections. This dataflow graph structure, along with the information associated with each processing element, is given to the real-time analysis tool. An overview of the modeling procedure is shown in Figure 8.

The following sections describe the process of converting the dataflow graph into an equivalent finite state machine representation. Once the system can be described as a FSM, algorithms can be used to explore this state diagram to gain timing information about the system.

Building the State Machine Representation

The dataflow representation from the model is transformed into a finite state machine representation of the system. This equivalent FSM representation of the system is constructed from the dataflow description of the system by “simulating” the execution of the system. The goal of this simulation is to explore all possible ways the system may be executed and build a FSM graph to store this information. It will be shown how this exhaustive search is performed efficiently on the FSM.

To begin this simulation the node that is marked as being the periodic process is executed (in the simulation). The data that this process produces triggers other processes to be put into the process queue. The process with the highest priority that meets its execution constraints (IF_ALL/IF_ANY requirements) is removed from the queue and executed. This continues until the process queue is empty and no other processes are to be run.

Branching in the execution occurs when the conditional outputs are encountered and when two processes of the same priority are ready to run at the same time. At these points a new execution path is created. These new paths are serviced in a depth-first recursive manner.

From this simulation of the dataflow system execution a state machine is built. As each process is executed a transition is added to the next state equation of the FSM. This transition from process A to process B in the FSM tells us that it is *possible* that process B will execute after process A.

An integer number is associated with every transition that is added to the FSM. This number represents the execution time of the process of which the transition is transitioning from. This augmented FSM graph provides a way to represent all possible ways a system may execute.

Processes are allowed to have the same priority. In this case the execution is simulated such that a case occurs where process A runs first then process B and another case where process B runs first (in terms of the state diagram it is split at this point into two branches). If a process has a conditional output then the execution must simulate both cases. One case where the output produces data that triggers the execution of the next process and the case that no output is produced (therefore this process does not cause any other processes to start). This will also cause a branching of the state diagram.

Since our goal is verification of systems it is crucial that all possible ways a systems may execute be taken into consideration when building the FSM. We are not looking for a graph that represents the normal or average operation of a system, but rather a graph that includes all possibilities for execution including the unlikely cases that may normally go unnoticed.

It is clear that for systems that have processes that may or may not start other processes based on their input data a very large number of possible execution sequences can be produced. This rapidly increasing search space can quickly become too large to analyze. Using a OBDD representation of the state machine provides the means to efficiently search this state space. The size of the OBDD representation is kept to a manageable size as the state machine grows exponentially larger. There is no need to actually create the state machine structure. A next-state equation is used to completely represent the state machine. The transitions are incrementally added

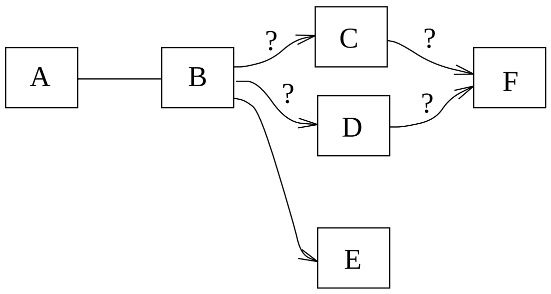


Figure 9: Example Dataflow graph

the this next-state equation. Since this next-state equation is simply a boolean equation it can be represented as an OBDD. This helps avoid state-explosion problems since the OBDD is usually a much more compact representation.

FSM System Representation Problems

As explained in the previous section, the dataflow representation of the system is converted to a finite state machine representation. In some cases the finite state machine representation is not as simple as it would seem. Many times extra states may have to be added as we will see. Consider the following example in Figure 9. We have a dataflow graph which has six processes. The priority of the processes is in order A, B, C, D, E, F, where A has the highest priority and F has the lowest. The question marks on the connections indicate that the output is a conditional output. Process B has three output ports. Two of these ports are conditional; they may or may not output data. Processes C and D also have their single output port to be conditional. The other processes all have unconditional outputs which always output data each time the process is executed.

An attempt to draw a state machine representation is shown in Figure 10. Six

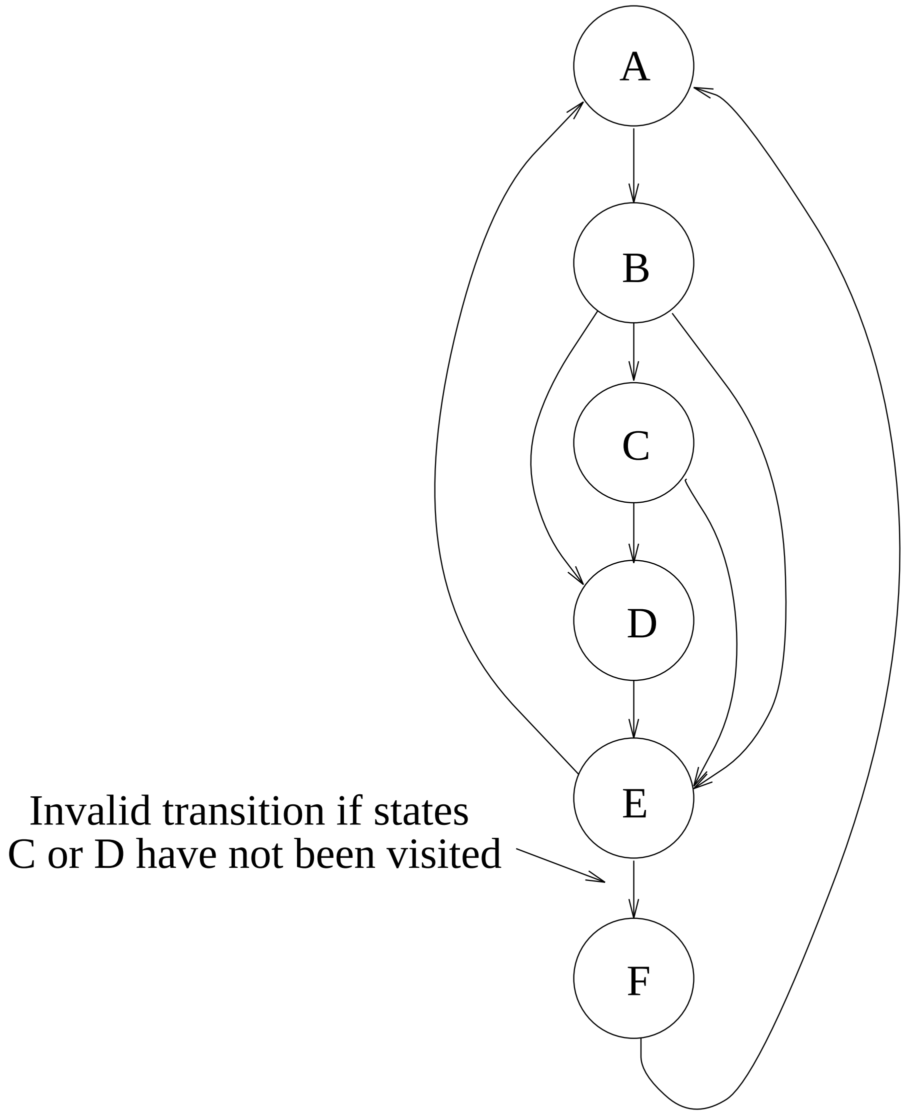


Figure 10: Problem representing system with FSM

states are drawn for the six processes and transitions are added. For example, process A always causes process B to execute. Process B always causes process E to execute and may cause process C and/or D to execute also. Transitions are added to represent this. Process F may only execute if processes C or D have executed. However, our graph has a path from A to B to E to F. The problem is that the transition from E to F is clearly invalid since it is shown to be able to occur without processes C or D firing. The transition was not placed there incorrectly.

Our current state machine with six states has no way to represent this piece of information. How can this be resolved? Either the FSM definition could be modified to include a new type of transition or extra states could be added. We do not want to change the way the FSM operates. This would cause the FSM to become more difficult to traverse. The latter option was chosen. As shown in Figure 11, the process E state was split into two states. This state machine represents correctly that an execution sequence A - B - E - F may not occur. This problem is a result of allowing conditional outputs on the processes. This gives a behavior of the process that is not inherently modeled by a FSM. As a result, in some cases states may need to be split into multiple states to model the operation of the system correctly.

When the state machine representation is constructed from the dataflow graph these situations must be taken care of. When the dataflow graph is simulated if a branching occurs the program recursively starts a new object to handle these new execution directions. Each of these execution sequences executes until it has no more processes in its queue. To solve this problem described above, each execution sequence does not add his transitions to the FSM until it is finished. When the transitions are

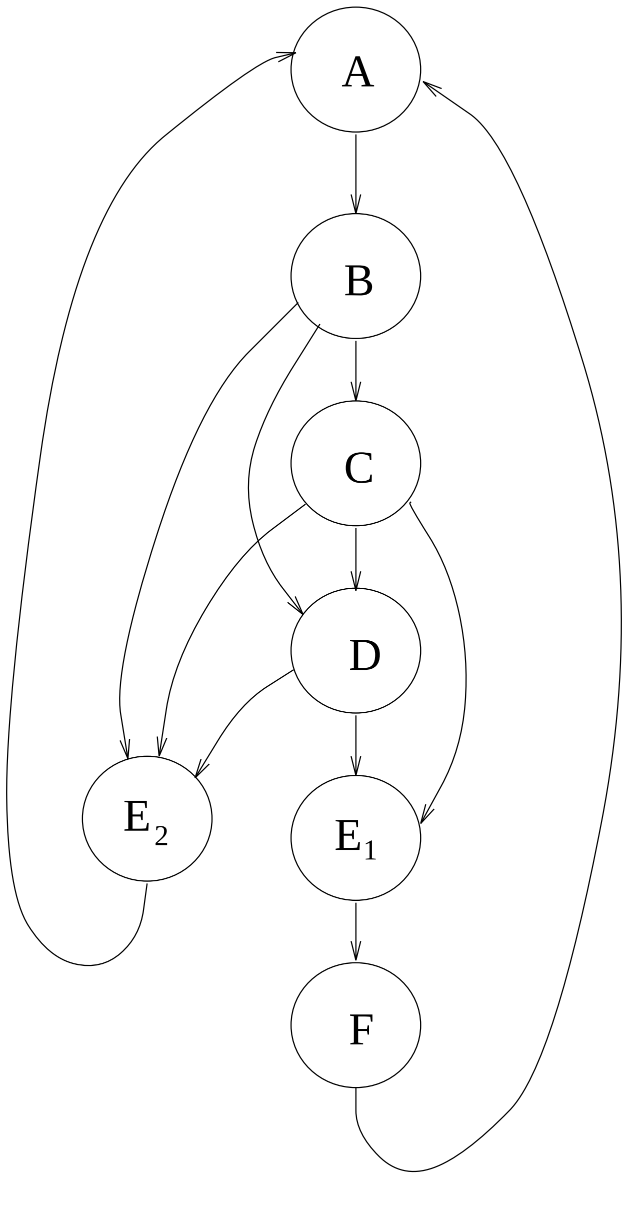


Figure 11: Corrected State Machine

added to the FSM care is taken not to put in invalid transitions. If necessary, new states are added to produce a correct FSM.

We now have a state machine representation of our real-time system. This state machine contains all possible ways the system may execute according to the information given in our model. This FSM is represented by a single boolean function, its next state equation. This equation is represented symbolically by an OBDD. Using the functions developed to find the next state of a given state or the previous state of a given state we have a way to move around within our state diagram. These functions work equally well on state sets as well as individual states. These functions provide a basis for the following algorithms that are used to determine the timing characteristics of a system.

BDD Algorithms Upper-Lower

Two algorithms were developed for finding the longest and shortest time between state sets. The state machine's next-state equation representing the state space is passed to these functions in the form of an OBDD. [3] Two state sets, an initial state set and an ending state set, are passed to the function each in the form of an OBDD. These functions return an integer specifying the longest or shortest time between these two state sets.

The Lower bound algorithm steps through the state diagram using the next state equation. It uses the FindNextState function to step forward from a current state set to its next state set. This algorithm starts with an initial state set and keeps a current state set. At each step it looks for an intersection of the current state set and the ending state set. A non-null intersection set means we have reached a ending state. The time that has elapsed since the beginning state is returned.

Upper bound algorithm works in a similar fashion. It steps through the state diagram beginning with an initial state set and maintains a current state set at each step. This algorithm traverses the state space noting when an intersection of the current state set and the ending state set occurs. The maximum time from the initial state set until the ending state set is returned as the longest path between the two given state sets. These algorithms are shown below.

```
int lower(BDD initial, BDD final, BDD N)
{
    int i=0;
    BDD R, Rp;
    R = initial;
    Rp = FindNextState(R,N) + R;
    while ( (Rp != R) and ((R and final) == 0) )
    {
        i++;
        R = Rp;
        Rp = FindNextState(Rp,N) + Rp;
    }
    if( (R and final) != null)  // if R and final intersect
        return i;
    else
        return INF;
}
```

```
int upper(BDD initial, BDD final, BDD N)

{

    int i=0;

    BDD R, Rp, Final;

    Final = final;

    R = TRUE;

    Rp = !Final;

    while ( (Rp != R) and ((Rp and initial) != 0) )

    {

        i++;

        R = Rp;

        Rp = FindPrevState(Rp,N) and !Final;

    }

    if (R == Rp)

        return INF;

    else

        return i;

}
```

Algorithm Implementation

The upper and lower bound algorithms described above are for unit-length transition FSMs. In this application we want each state to take a specified length of time to execute to represent the time it takes for a process to execute. As discussed earlier, we could simply add states to our state machine to show the non-unit length transitions. This is done by adding n number of states in series to show that a transition takes n units of time to take place as shown in Figure 12. For systems with long transition times this is not feasible. It is desirable to use a FSM with times associated with each state. This method provides a compact representation but, as we will see, it does add complexity to the execution of the FSM.

Although the modified state diagram is a compact representation of the real-time system, it is not a straightforward task to traverse its state space in a breadth first search manner as we want. Two ways were examined to execute the modified state diagram.

One way is to create another state machine to represent the modified state machine. This state machine would be easier to execute than the modified one and thus, easier to apply the upper and lower bound algorithms to explore the state space. This is shown in Figure 13. This state machine basically steps through the FSM in time (relative to the initial state) and creates this new, expanded state machine. This state machine is essentially state transitions to sets of states.

This new state machine does, however, require more state bits to encode the new transition lengths. The advantage of this method is that the upper and lower bound functions remain relatively simple. The OBDD operations on this state graph

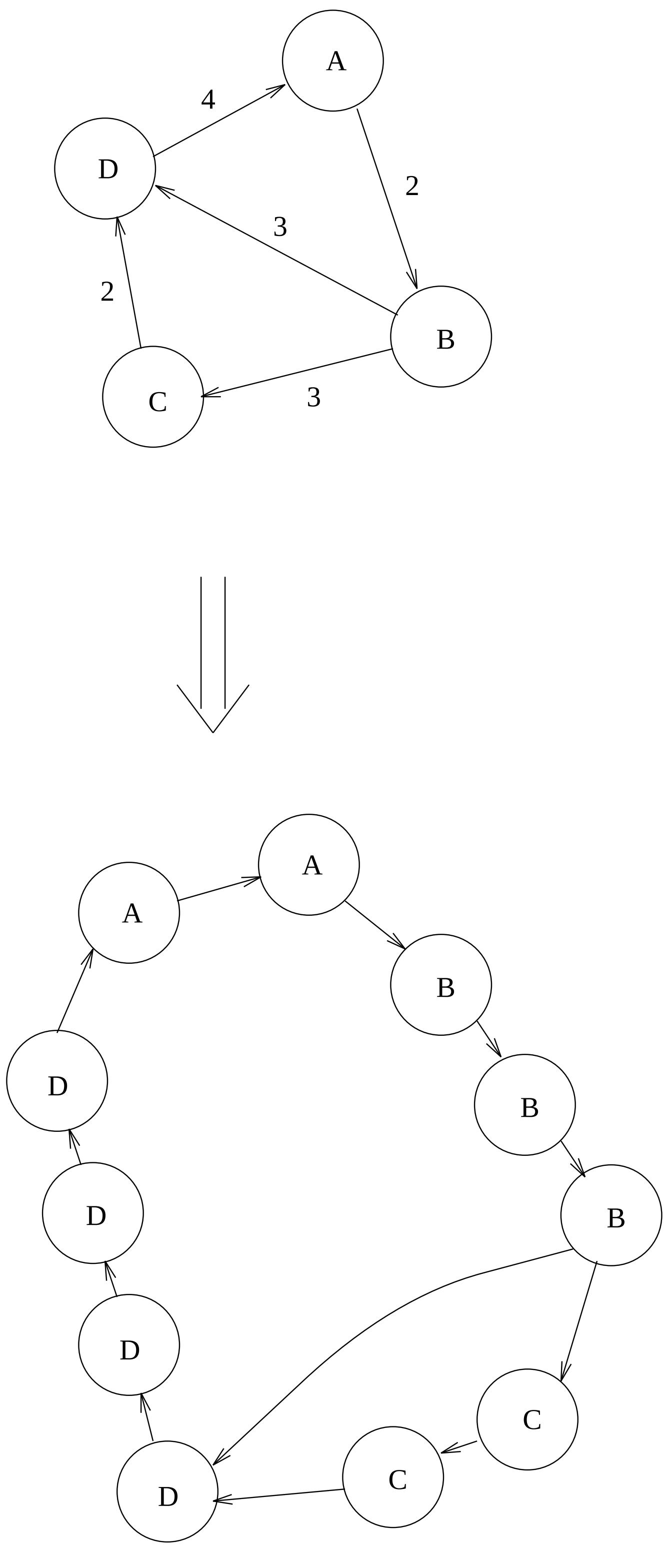


Figure 12: Implementations of a timed transition FSM

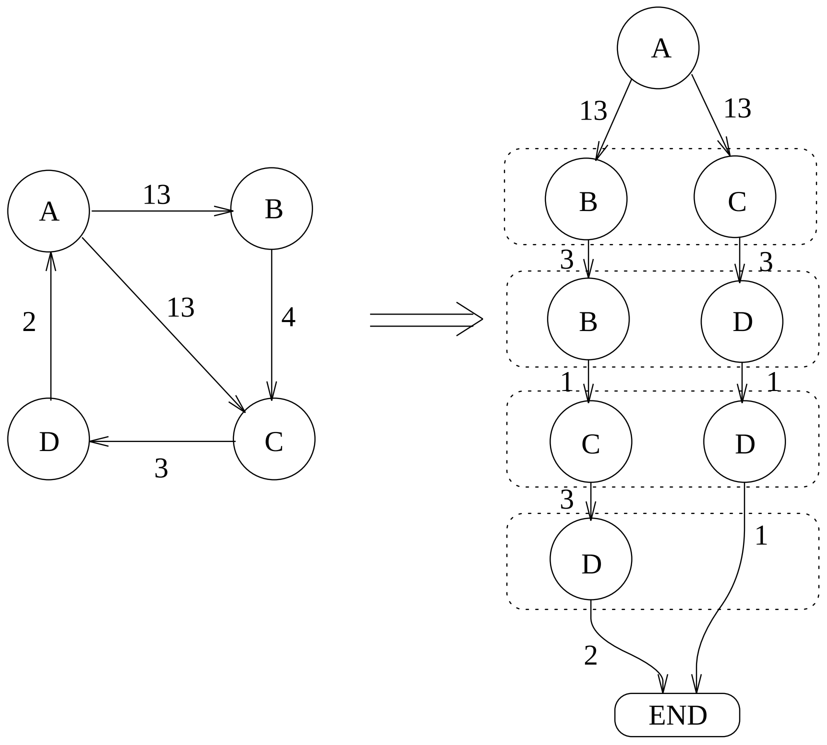


Figure 13: Another representation of Modified FSM

become somewhat slower because of the increased number of variables to represent the increased number of state bits. This will be partially offset by the fact that the verification algorithms can remain essentially the same as they are for the unit-transition algorithms.

Another method is to execute the modified state machine in the analysis algorithms themselves. This allows us to use our modified state machine representation directly. It does, however, add much more complexity to the analysis algorithms. The same idea as above applies to this approach. The state space is stepped through in a breadth first search manner in order of length of time from the initial state. Each time transition is broken into smaller steps to implement the traversal. This method has the advantage of begin easier to return bad schedules (the execution paths that are past the deadline).

Verification Process

Our Real-Time system is now modeled as a Timed Transition Finite State Machine in the form of an OBDD. Using the above algorithms, we can find the longest and shortest time that a system may begin or finish executing within the frame cycle relative to the initial state.

The upper bound algorithm is used to find the longest path (time) between the initial state and every other process in the system. The maximum is found of these times. This is the maximum time it will take for all processes to finish execution. If this time is greater than the deadline given for all processes to finish execution then there is at least one case where the system may not meet its deadline.

If requested, schedules that run past the deadline can be returned. This provides the designer with the exact sequence of execution of processes that will cause the system to fail.

This recording of individual execution paths does require extra storage in memory, however. A path for each separate execution possibility must be stored. It is stored as a linked list of small BDD items as shown in Figure 14. At each step in the traversal of the state diagram the path is recorded. New objects are created containing a OBDD representing a single state and a pointer to the previous object in the sequence.

If the longest path between the two states is greater than the maximum allowed given then the schedules of those paths exceeding the maximum are returned. The paths that we are concerned with finding bad schedules for are usually paths that

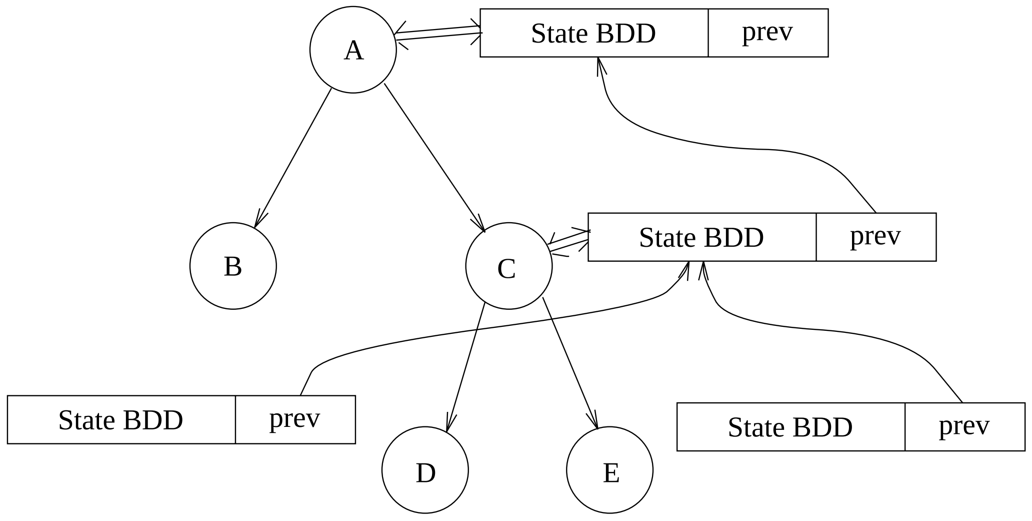


Figure 14: Keeping track of bad schedules

begin with the state that starts the frame period, the initial (or periodic) state.

This method of system verification builds on the the method of searching the state transition diagram using OBDDs as described by Campos [3]. In this work the model must be specified as a state-transition graph.

This method uses a model-based verification approach. Unlike the work of Campos, in this approach the model of the real-time system is entered graphically. This graphical representation of the models is transformed into the finite state machine model. This conversion process is fully automated. A large part of the problem is modeling the system as a finite state machine and avoiding state explosion problems as this state machine is begin built. In many systems the mapping is very complex depending on the type of scheduling algorithms used, the constrains placed on the execution of the processes, etc. Since OBDDs are used to represent the state machine it may be built up incrementally. This approach is demonstrated in the example in the following section.

CHAPTER IV

EXAMPLE

Data Acquisition Example

The following example is of a data acquisition system to monitor plant conditions of a jet engine test. This system has one periodic process which is the *acquire* process. This process executes every 13ms. It reads data from the various external sensors on the engine and plant. The data collected by the *acquire* process is sent to the *preprocess* process which triggers its execution. The *preprocess* process provides some filtering of erroneous data. The *preprocess* process, based on its results, decides whether or not to start the error detection processes (*pressure* or *temperature*). The *preprocess* process may start any combination of these. If any of the detection processes decide that a problem may exist they can signal a warning. Each of these detection processes make this decision individually, based on the data given to them, whether or not to start the *warning* process. If the *warning* process is executed it always triggers the *yellow* process which announces this warning condition. The *warning* process can, based on the data, start an *alarm* process. This *alarm* process will always start a *red* process which announces the alarm condition exists.

The *preprocess* process also always sends its results to the *display* and *store* processes. The *display* process is used to display data on some type of output. The *store* process is used to store the data on some type of permanent media for later use. The dataflow diagram is shown in Figure 15.

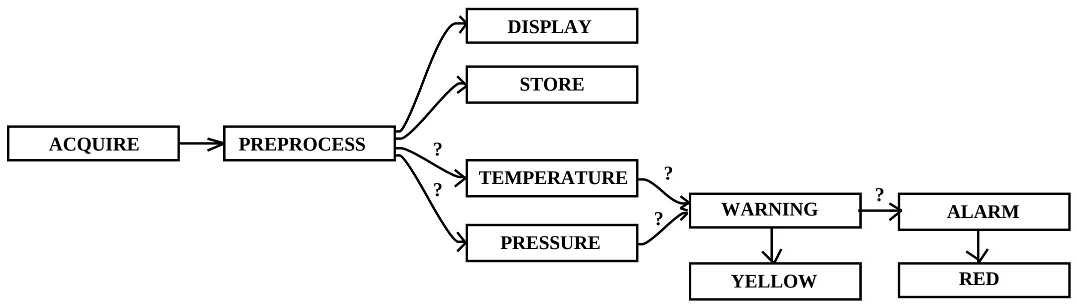


Figure 15: Dataflow diagram for Data Acquisition Example

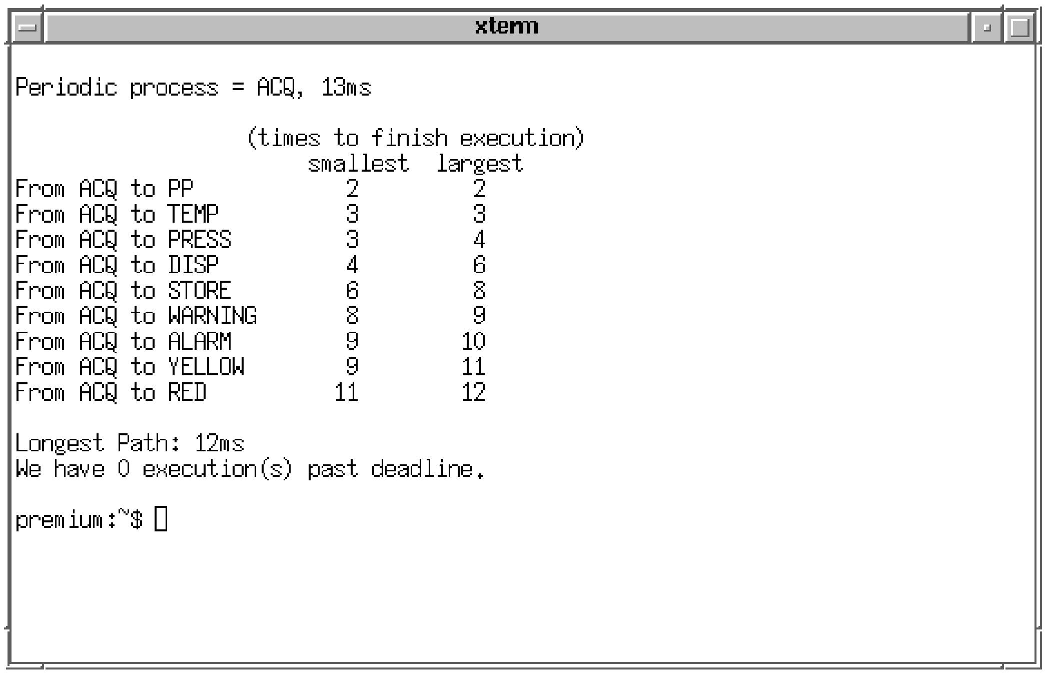
All processes execute for 1ms except the *display* and *store* processes. These two processes execute for 2ms. The priority order from highest to lowest is as follows: *acquire, preprocess, temperature, pressure display, store, warning, alarm, yellow*, and *red*.

The *acquire* process is executed every 13ms which starts the cycle. The requirement for schedulability is for all processes to finish execution before the next execution of acquire begins. The upper and lower bound algorithms can be used to find the the shortest and longest times between the periodic *acquire* process and each of the other processes in the system. If the longest time for each process to finish execution is less than the data acquisition period then no process can overlap into the next period and the system is schedulable.

In this example the longest possible time for a process to finish execution was *red* which could finish as long as 12ms after the period began. The output of the analysis program is shown in Figure 16. The output shows the longest and shortest time it may take to execute between two processes. These times are shown between the periodic process, *acquire*, and each of the other processes in the system. By looking at this chart we can get a good idea of how the system will perform. The output shows that the system is schedulable in all cases. It has at least 1ms to spare in the worst case and normally much less that that since it is rare that both sensors will signal an alarm at the same time.

Suppose someone suggests that the *warning* and *yellow* processes should have a higher priority to provide a faster warning response time. The relative priorities are changed to give the warning and yellow processes a higher priority as follows: *acquire, preprocess, warning, yellow, temperature, pressure, display, store, alarm*, and *red.* The upper and lower bounds algorithms were again used to compute the maximum and minimum times for each process to complete execution. The results of the analysis are shown in Figure 17.

By looking at the output we can clearly see that the system is not schedulable in all cases. The worst case sequence now takes 14ms. The schedule that extended past the deadline is as follows: *acquire, preprocess, temperature, warning, yellow, pressure, warning, yellow, display, store, alarm*, and *red.* The cause is now obvious. Raising the *warning* process's priority causes it to execute once for every time it is requested by each detection process. In the original process assignment if one or both processes requested a warning it was only executed once since processes are not allowed to be in the scheduler's queue multiple times.



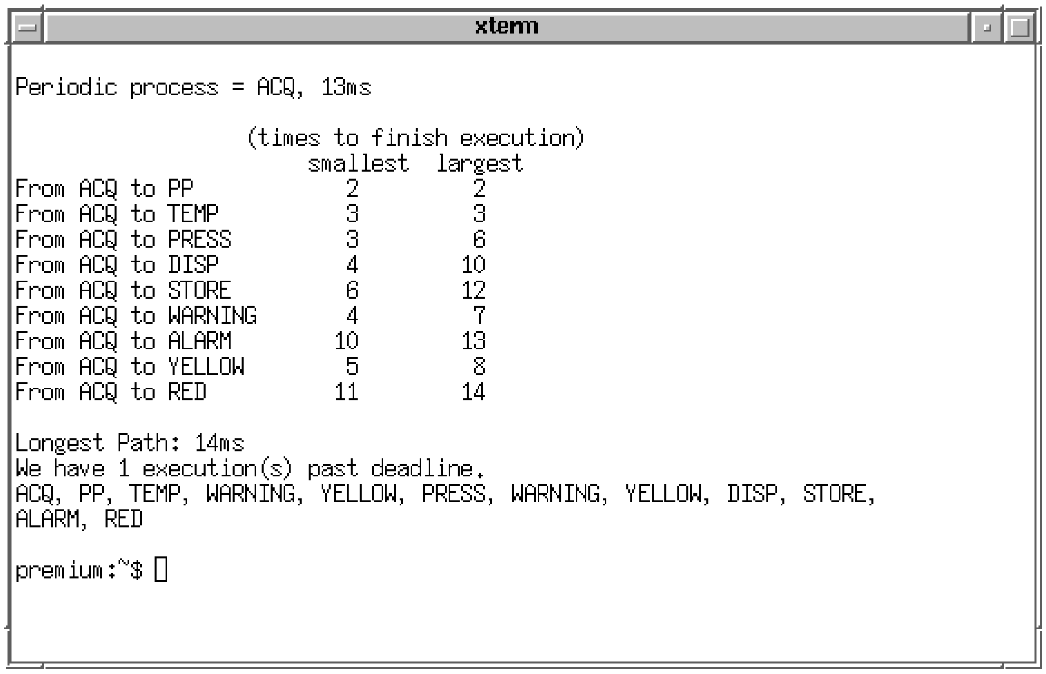
```
Periodic process = ACQ, 13ms

                  (times to finish execution)
                       smallest  largest
From ACQ to PP            2         2
From ACQ to TEMP          3         3
From ACQ to PRESS         3         4
From ACQ to DISP          4         6
From ACQ to STORE         6         8
From ACQ to WARNING       8         9
From ACQ to ALARM         9        10
From ACQ to YELLOW        9        11
From ACQ to RED          11        12

Longest Path: 12ms
We have 0 execution(s) past deadline.

premium:~$
```

Figure 16: Output of analysis tool - Case I



```
Periodic process = ACQ, 13ms

                  (times to finish execution)
                       smallest  largest
From ACQ to PP            2         2
From ACQ to TEMP          3         3
From ACQ to PRESS         3         6
From ACQ to DISP          4        10
From ACQ to STORE         6        12
From ACQ to WARNING       4         7
From ACQ to ALARM        10        13
From ACQ to YELLOW        5         8
From ACQ to RED          11        14

Longest Path: 14ms
We have 1 execution(s) past deadline.
ACQ, PP, TEMP, WARNING, YELLOW, PRESS, WARNING, YELLOW, DISP, STORE,
ALARM, RED

premium:~$
```

Figure 17: Output of analysis tool - Case II

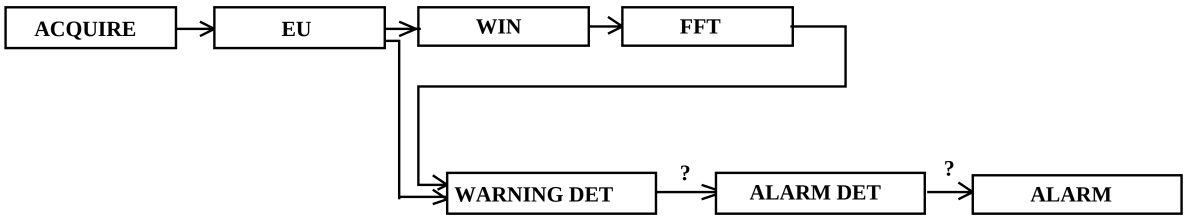


Figure 18: Dataflow graph of signal processing example

Signal Processing Example

The following example shows a signal processing system. The system has a periodic process *acquire* which gets an input sample block of data. There is a *EU* process that performs engineering unit conversion on all input data to scale it to proper values. The *win* process applies a window function to the data before it goes into the FFT routine. The next process is the *fft* routine. This routine performs a Fast Fourier Transform on the data set. The output of this process is data that is now in the frequency domain. The output of the fft process goes into the *warning/detection* process. This process looks for abnormal spectral information that could signal a problem. The output of this process is modeled as a conditional output. This means the process may or may not output data. The output of this process (if there is output) feeds into the *alarm/detection* process. Its other input gets EU converted sample data from the EU process. This process is a IF_ALL process. Data must be present on both these inputs before it can be available for execution. The output of the process is also a conditional output. It feeds into the *alarm* process. This process's job is to actually signal an alarm if needed (if data is given to it). The dataflow graph of this model is shown in Figure 18.

The example was entered into the model editor as shown in Figure 19. Each process was given a execution time and assigned a priority. The priority order from highest to lowest is as follows: *acquire, EU, win, FFT, warning/detection, alarm/detection*, and *alarm*.

The processes *acquire, EU*, and *win* each execute for 1ms. The *FFT* process executes for 5ms. The *warning/detection, alarm/detection*, and *alarm* processes execute for 2ms. The priority order from highest to lowest is as follows: *acquire, preprocess, temperature, pressure display, store, warning, alarm, yellow*, and *red.* The *acquire* process executes periodically at every 15ms.

The interpreter/analysis tool was executed with this model as its input. The output was as shown in Figure 20. The output shows the same largest and smallest distances between the period process and every other process.

This is somewhat unexpected, but correct. The conditional outputs in this case only cause the execution sequence to end earlier. If the *warning/detection* process does not output data then, for this example, no more processes are scheduled to be run and the current period's processing is finished. The same is true if the *alarm/detection* process does not output data. If this processes input discipline was IF_ANY then this would not be the case since the *EU* process's data output would schedule both the *WIN* process and the *alarm/detection* process.

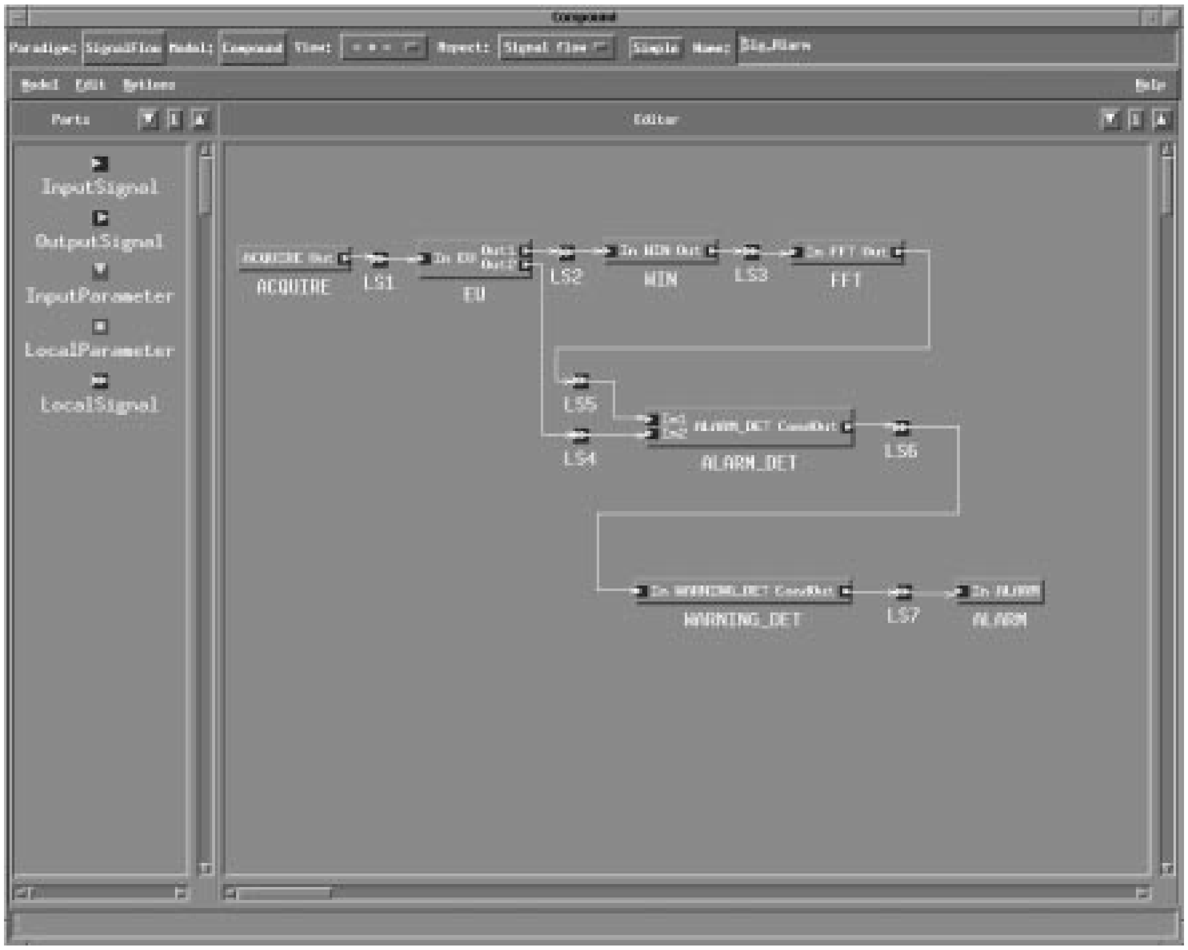
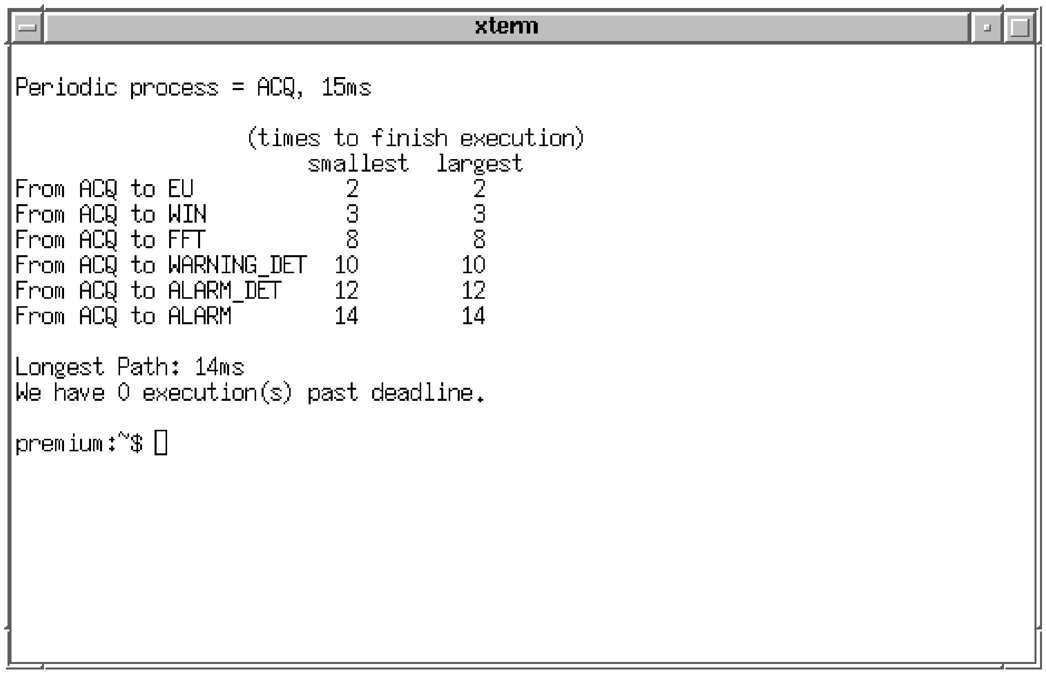

Paradigm: SignalFlow Model: Compound Aspect: Signal flow Simple Name: Sig_Alarm

Figure 19: Signal processing example in Model Editor



```
Periodic process = ACQ, 15ms

                  (times to finish execution)
                       smallest  largest
From ACQ to EU            2         2
From ACQ to WIN           3         3
From ACQ to FFT           8         8
From ACQ to WARNING_DET  10        10
From ACQ to ALARM_DET    12        12
From ACQ to ALARM        14        14

Longest Path: 14ms
We have 0 execution(s) past deadline.

premium:~$ 
```

Figure 20: Signal processing example output

CHAPTER V

CONCLUSIONS

It is necessary to verify real-time systems in critical applications. Many times people do not throughly test real-time systems. A rare series of events may occur at the worst possible time to produce system failure. It is generally not possible to fully test an actual system through enumeration of all execution possibilities. The number is simply too large.

Scheduling algorithms are available that provide an optimal scheduling solution but cannot guarantee schedulability in all cases. Model-based analysis can provide a way to explore the state space of a system. For many systems this is a difficult task since state space of the system grows exponentially as the complexity increases. A way of examining this space efficiently is needed.

This application demonstrates the advantages of using OBDDs for representing a problem domain in terms of a finite state machine. The full enumeration of the problem space never needs to be constructed thereby avoiding the state explosion problem.

A model-based analysis method using OBDDs is well suited to this task. From the system dataflow model a finite state machine representation is derived. Algorithms that can determine schedulability based on the FSM have been developed. The ability of the algorithms to analyze a large state space efficiently is made possible by using OBDDs. They provide an good representation of the finite state machine which can be explored on a state set basis. Finding a next state set or previous state set from a current state set given a next state equation is a straightforward task using the OBDDs.

This tool not only determines schedulability of a system, but can return the bad schedules which cause a system to miss its deadline. This information can help the systems designer be aware of potential problems even before the system is implemented.

Model-based analysis can give the system designer information about the system's timing characteristics while the system is still in the design stages. The designer can also perform the analysis on different "what if" situations by simply changing the model.

Much experience was gained using OBDDs. The OBDDs were implemented using a C++ OBDD package developed at Vanderbilt. Using this OBDD implementation provided testing and feedback for the package. Building this application gave insight into how efficient OBDDs are and how large of a state space that can be traversed. The example shown previously uses a state encoding of 30 bits and executed with short computation times on the order of a few minutes. (The OBDD actually used has 60 variables to represent the characteristic next state equation of the state machine). This gives a possible state space of up to 2^{30} states that can be represented efficiently since the search times have been found to be largely dependent on the number of variables used in the OBDD.

Future Research

Future directions include extending this method to verify real-time parallel processing systems where the data flow between processors must be modeled. This type of analysis would be valuable in Multigraph applications in which the actor nodes are spread across both distributed or shared memory multiprocessors.

The method could also be extended to systems with multiple cyclic processes on a single processor. This would enable modeling of systems with more than one periodic process such as a data acquisition systems which sample data at more than sample rate.

Also, a more detailed analysis could be performed if a upper and lower bound were assigned to each processes execution time instead of a single time. The process would be defined to execute for this nondeterministic amount of time.

Appendix A

MODEL DEFINITION FILE

```
attribute Datatype : menu "Select datatype"
                        { "Buffer" Buffer_type;
                          "Int"    Int_type ;
                          "Float"  Float_type ;
                          "Double" Double_type ;
                        };

attribute Description : page "Model description:" (2 64) "";

paradigm SignalFlow {
   classes
      atom Signal {
         attr Datatype;
       };
      model Processing {
        SignalFlowAspect {
            attrs {
                attr Description;
            }
         parts {

            InputSignals    : InputSignal link  ;

            OutputSignals   : OutputSignal link ;

          }

     }

   }

atoms

   InputSignal "isig.icon"  is_a ( Signal );

   OutputSignal "osig.icon" is_a ( Signal ) {

     Outtype : menu "Data Output when executed"

                    { "Always" Always_out;

                      "Conditional" Cond_out;

                    };

   };

   LocalSignal "lsig.icon"  is_a ( Signal ) {

     QLength : field int "Queue length:" "10";

   };

models

   Primitive is_a ( Processing ) primitive {

     SignalFlowAspect "Signal flow" {

         icon rect {

             left : InputSignals;

             right: OutputSignals;
            top:   InputParameters;

        };

        font 3;

        color foreground;

        attrs {

            Script : page "Script name/Timer delay:" (1 64) "";

            Time     : field int "Time:" "1";

            Priority : field int "Priority:" "10";

            Period   : field int "Period:" "0";

            Kind : menu "Select primitive kind"

                        { "Algorithm" AlgorithmicPrim ;

                          "Periodic"  PeriodicPrim;

                        };

            Discipline : menu "Select control discipline"

                        { "IfAll" If_All ;

                          "IfAny" If_Any ;

                        };

        }

  }

}

Compound is_a ( Processing ) compound {

  SignalFlowAspect "Signal flow" {

        icon rect {
```

```
        left : InputSignals;

        right: OutputSignals;

    };

    font 3;

    color foreground;

    attrs {}

    conns {

       DataflowConn { 1 solid line arrow } :

          { InputSignals -> PrimitiveParts InputSignals }

          { InputSignals -> CompoundParts  InputSignals }

          { LocalSignals -> PrimitiveParts InputSignals }

          { LocalSignals -> CompoundParts  InputSignals }

          { PrimitiveParts OutputSignals -> LocalSignals }

          { CompoundParts  OutputSignals -> LocalSignals }

          { PrimitiveParts OutputSignals -> OutputSignals }

          { CompoundParts  OutputSignals -> OutputSignals };

    }

    parts {

       LocalSignals  : LocalSignal;

       PrimitiveParts: Primitive hierarchy;

       CompoundParts : Compound hierarchy;

    }

}
```

```
        }
}
```

The Model Definition File (MDF) file specifies the modeling environment. The Model Editor is custom built according to the specifications in the MDF. The MDF is written in is own declarative language to describe the modeling environment. The MDF file shown here was used to build the Model Editor used in this application to enter the models for analysis.

The MDF file begins with attribute definitions. These attributes are defined at the beginning for use later on in describing options that can be added to other objects. The attributes that are predefined in this file are a menu to select the data type of a object and a text entry field.

This model uses the *signal flow* paradigm. The paradigm consists of 3 main sections: class definitions, atom definitions, and model definitions. For this signal flow type modeling environment 2 class elements are defined: a *Signal* element and a *Processing* element. Later on in the definition file other objects will be derived from these classes.

Three atoms are defined for use in this environment: InputSignals, OutputSignals, and LocalSignals. Each of these atoms are derived from the Signal atom defined earlier. Each also have a bitmap file for assigning a icon to them to define how they will look in the model editor. In this application the icons are small boxes with arrows pointing in the proper direction.

The OutputSignal icon has a menu attached to it so that the user can select individual outputs to be conditional or unconditional. In the editor the user must click on the OutputSignal icon to bring up this menu.

This signal flow paradigm is defined to use 2 kinds of models: *primitive* models and *compound* models. The primitive model has several attributes specified:

- Script Name - Name of process the primitive represents
- Time - Length of time it takes the process to execute
- Priority - Priority of process
- Period - Period at which process executes. This is only used when the *kind* (below) is "Periodic".
- Kind - Algorithmic or Periodic: Distinguishes the period node. Only one should be selected for a system. All other processes should be algorithmic.
- Discipline - IF_ALL / IF_ANY: Specifies whether a process's must have data present on all his input ports before it can be available for execution or if data only has to be available on any of its input ports to be able to run.

The compound models defined here do not have any attributes. These models contain other models, compounds or primitives, and their connections. Valid objects that may be placed in a compound are listed under "parts". For this model they are: LocalSignals, Primitives, and Compounds. Valid connections that may be made between these objects are specified in the MDF under "conns".

The MDF file provides a precise method of specifying the modeling environment. It is not just used as a specification but is actually what is used to custom build the modeling environment.

REFERENCES

[1] Abbott, B., Bapty, T., Biegl, C., Karsai, G., Sztipanovits, J.: "Model-Based Approach for Software Synthesis," IEEE Software, pp. 42-53, May 1993.

[2] Bryant, R.: "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", School of Computer Science, Carnegie Mellon University, July 1992, CMU-CS-92-160

[3] Campos, S., Clarke, E., Marrero W., Minea M.: "Timing Analysis of Industrial Real-Time Systems", pages 97-107, Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.

[4] Campos, S., Clarke, E.: "Real-Time Symbolic Model Checking for Discrete Time Models", School of Computer Science, Carnegie Mellon University, May 1994, CMU-CS-94-146

[5] Liu, C., Layland, J.: "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of the ACM, 20(1), January 1973.

[6] Campos, S., Clarke, E., Marrero, W., Minea, M.: "Computing Quantitative Characteristics of Finite-State Real-Time Systems",School of Computer Science, Carnegie Mellon University, May 1994, CMU-CS-94-147

[7] Bapty, T., Abbott, B.: "Portable Kernel for High-Level Synthesis of Complex DSP-Systems," *Proceedings of the International Conference on Signal Processing Applications and Technology*, Boston MA, 1995.

[8] Sztipanovits, J., Karsai, G., Biegl, C., Bapty, T., Ledeczi, A., Misra, A.: "MULTIGRAPH: An Architecture for Model-Integrated Computing," *Proceedings of the International Conference on Engineering of Computer Systems*, Ft. Lauderdale, Fla., October 1995.

[9] Karsai,G.: "A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming," *IEEE Computer*, pp. 36-44., March 1995.

[10] Stankovic, J., Spuri, M., Di Natale, M., Buttazzo, G.: "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Computer*, pp. 16-25., June 1995.

[11] Bapty, T., Abbott, B.: "Parallel Signal Processing for Turbine Engine Testing," Final Report for USAF-UES SRP, Contract no. F49620-88-C-0053, July 22, 1991.

[12] Moore, M., Karsai, G., Sztipanovits, J.: "Model-based programming for parallel image processing," *Proc. of the 1st IEEE International Conference on Image Processing*, 1994.

[13] Abbott, B., Bapty, T., Biegl, C., Karsai, G., Sztipanovits, J.: "Model-Based Approach for Software Synthesis," *IEEE Software*, pp. 42-53, May, 1993.

[14] Ledeczi, A., Bapty, T., Karsai, G., Sztipanovits, J.: "Modeling Paradigm for Parallel Signal Processing," *The Australian Computer Journal*, vol. 27, No. 3, pp. 92-102, August, 1995.

[15] Levi, S., Agrawala, A.: *Real Time System Design*, New York: McGraw Hill, Inc., 1990, pp. 169-172.

[16] Gomaa, H.: *Software Design Methods for Concurrent and Real-Time Systems*, Reading, Massachusetts: Addison-Wesley Publishing Company, 1993, pp. 5-6, 123-132.

[17] Parks, T., Lee, E.: "Non-Preemptive Real-Time Scheduling of Dataflow Systems," Presented at *IEEE International Conference on Acoustics, Speech, and Signal Processing*, May 1995.

[18] Hopcroft, J., Ullman, J.: *Introduction to Automata Theory, Languages, and Computation*, Reading, Massachusetts: Addison-Wesley Publishing Company, 1979, pp. 13-15.

ELECTRICAL ENGINEERING

A METHOD FOR MODELING AND VERIFICATION OF REAL-TIME SYSTEMS

JASON MATTHEW SCOTT

Thesis under the direction of Professor Gabor Karsai

Verification of real-time systems is essential. Presented here is a method for modeling real-time systems and computing the model's timing characteristics automatically. A model-based approach allows the system to be designed and analyzed before it is actually constructed.

The real-time systems that are considered in this method are systems that can modeled as a data-driven system. The model of the system is a dataflow graph. The system has one periodic process. From the dataflow model an equivalent finite state machine representation of the system is produced by this package. To provide efficient traversal of this large state space an Ordered Binary Decision Diagram (OBDD) is used to represent the state machine using symbolic methods. Algorithms have been developed to find the largest and smallest distances (times) between any two state sets. From these algorithms schedulability of the real-time system can be determined. If the system is found to have a case where it may miss its deadline then the sequence of events that may lead up to this event can be given to the user.

Approved____________________________ Date____________________
Advisor