EFFICIENT VERIFICATION OF MULTI-PROCESSOR REAL-TIME SYSTEMS USING SYMBOLIC METHODS

By

Jason M. Scott

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

August, 2003

Nashville, Tennessee

Approved: Date:

ACKNOWLEDGEMENTS

Most of all I want to thank my wife, Amy, for her constant support and motivation. I would not have continued this dissertation if it were not for her support. She had to endure many nights alone so that I could complete this work and I cannot thank her enough. I would like to thank my family for their encouragement and for always being there when I needed them.

I am very appreciative of my advisor, Dr. Gabor Karsai, for his advice, technical feedback and encouragement (and much patience!). I would like to thank all the people who have helped me during my time as a graduate student at Vanderbilt. ISIS is composed of a friendly and knowledgeable group of people who have been great to work with. I would especially like to thank Dr. Ted Bapty for funding support and the interesting projects that I have had an opportunity to work on. I would also like to thank Dr. Richard Davis, Dr. Sandeep Neema, Dr. Greg Nordstrom and Brandon Eames for their friendship and encouragement. I have learned a lot from working with these guys in my time at Vanderbilt. Thanks also to the other members of my committee: Dr. Janos Sztipanovits, Dr. Vijay Raghavan, Dr. Bharat Bhuva, and Dr. Benoit Dawant.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER I

INTRODUCTION

Increasingly, especially in the last decade, embedded computer-based systems are used in nearly every aspect of life in the modern world. They typically consist of several processors interacting with each other and interacting with their environment. As of 1999, it has been reported that a fraction of 1% of the world's microprocessors are used in general-purpose computers [3]. The remaining 99% of microprocessors are used in embedded systems. Some of the most important computer systems we depend on today are embedded *real-time* systems. Real-time systems must not only produce correct results but also produce these results in a timely manner. Real-time systems have timing requirements that must be satisfied for the system to operate correctly. In safety-critical applications the consequences of failure of a real-time system to meet its timing constraints can mean disastrous results (financial/property loss, even loss of life).

Real-time embedded system development and implementation are complex tasks. Systems are likely multi-processor systems interacting with several input/output devices and executing hundreds of software tasks. Design automation tools are necessary for automation of tedious and error-prone tasks such as managing the task allocation and communication of such a complex system. Small changes in an application may necessitate many implementation changes that may propagate throughout the entire system. For example, the addition or deletion of a single task may necessitate the re-allocation of tasks to other processors in order to load-balance or communication-balance the system to meet real-time requirements. As embedded systems become more complex due to advances in technology and design automation, verification and analysis tools are a critical part of an overall design approach. It is essential to verify that the system design and generated implementation of that design satisfies system resource and performance constraints.

This dissertation is concerned with system applications in the signal-processing domain. These systems are described with large-grain macro-dataflow models implemented on systems with multiple processing elements (multi-processor systems). Dataflow is well suited for the description of signal processing systems because it is the natural way signal processing engineers specify their algorithms. A dataflow language has several advantages for the programming of multi-processor hardware. Concurrency is explicit in dataflow languages. No additional work is needed to decompose the algorithm into a parallel implementation. Also, the user is insulated from the processor task scheduling that must occur on each processing element. From the user's point of view, the dataflow graph execution must simply satisfy the dataflow semantics.

The dataflow implementation must be verified to be able to execute indefinitely for given input data rates, implying that the necessary size of all buffers must be bounded and, more specifically, fit within available system memory. The system must be proven to be free of deadlock. It must be verified that the system will *always* meet its end-to-end timing constraints. Verification must be performed to determine each of these requirements is satisfied in *all* possible system behaviors.

Although in some embedded system applications, it is possible to statically schedule the system, this is not always the case. Many systems require a dynamic, event-driven scheduler. Also, it is required that tasks not always be simple data transformation functions. Their behavior may be data dependant. It is not possible to statically schedule components whose behavior is only determined at run-time.

We would, however, like to perform a static analysis / system verification at the time of system generation, accounting for all possible system behavior variations that may occur at run-time. This set of all valid system states is known as the *state space* of the system. The state space of a system may be very large due to the combinatorial growth of the state space for a system with concurrently operating components. This is known as the *state explosion problem*. In this work, non-deterministic behavior is allowed so that behavior that may vary at runtime can be modeled. Non-deterministic behavior causes the number of possible system behaviors (*behavior space*) to grow dramatically, though not necessarily the number of system states.

Even for moderate size systems, the state space can grow to be extremely large. The size can be so large that it is not possible to represent the entire state space explicitly. Note that while it may be possible to represent the state space by creating the concurrent state spaces and maintaining them separately, it may not be possible to examine all possible system behaviors. For complete verification, the ability to traverse the state space examining all behaviors the system may exhibit is essential. Verification through simulation is not an option for such a large number of possible behaviors the system may exhibit due to non-deterministic behavior present in the model. A formal method known as symbolic model checking has had success in providing exhaustive verification of extremely large state spaces [4].

Model checking is a verification technique providing exhaustive examination of a system through the exhaustive exploration of a state space. *Symbolic model checking* provides exhaustive verification of a system by implicitly representing a state space through the use of a symbolic representation. It is called symbolic because it is based on the manipulation of Boolean formulas [1].

Frequently, symbolic model checking takes the form of a method for representing a finite-state transition system with the graph-based Boolean function representation *ordered binary decision diagrams* (OBDDs). OBDDs have been proven to work well in many cases for the representation of extremely large state spaces. OBDDs have had their greatest success in commercial hardware VLSI verification tools where very large designs must be managed efficiently [5]. Symbolic model checking uses a formal methods approach to system verification with the advantage that the users needs no special expertise in mathematical disciplines as is the case with using formal methods based on theorem proving.

The research presented in this dissertation focuses on the development of an analysis and verification system employing OBDD and a variation of OBDDs, known as MTBDD (Multi-Terminal Binary Decision Diagrams), based symbolic model checking techniques. The techniques resulting from this research will be used to verify schedulability, deadlock-free operation, and real-time timing constraints. The following statement represents the main objective of this research:

The goal of this research is to develop an efficient analysis and verification system for multi-processor, real-time systems using symbolic techniques.

A verification tool, Real-Time Analysis Tool (RTAT), has been developed implementing the model checking strategies for real-time systems presented in this dissertation.

The system model is a dataflow graph implemented on multiprocessor hardware. This model can be used for system synthesis. The verification tool operates on this same system model and can provide verification of both design and implementation correctness. In contrast, other widely used model checking tools require the user to provide a behavior model of system operation, usually created by hand from a system design description. If implementation verification is desired, the user must create a behavioral model of the system, usually abstracted by hand from a system implementation.

In this work, verification questions are posed in terms of *system* correctness. The use of other verification tools requires that the properties to be verified must be specified in terms of the underlying behavioral model of the system. These properties are usually specified in terms of a temporal logic, such as computational tree logic (CTL), that can be difficult for the user to express system level properties.

<u>The main contributions of this dissertation are:</u>

- An approach for the efficient modeling and analysis of *timed* finite-state systems is presented. Methods to analyze these models based on model checking techniques are developed using OBDDs and MTBDDs as the underlying representation of a finite-state behavior graph.
- A method for *exhaustive* verification of multi-processor real-time systems using a timed finite-state model-checking approach is presented. This dissertation tackles the problem of *schedulability* verification of a design and its implementation. Schedulability is defined as follows: "If a system is able to execute its dataflow graph continuously in all possible cases, then the system is schedulable." Also, *performance analysis* using the same underlying model checking approach is

presented which is capable of finding performance extremes, e.g., maximum throughput, is also explored.

Overview

Chapter II gives background information on formal verification, the symbolic representations of OBDDs and MTBDDs. Also given is background information on relevant model checking tools and current approaches to real-time system design focusing on scheduling approaches. Chapter III presents the general models of computation used in this work along with the variations tailored for use in this dissertation work. Symbolic methods utilized for the foundation of exhaustive system behavior analysis are detailed in Chapter IV. Chapter V presents a system verification method using a symbolic representation as the underlying computation engine. Chapter VI presents a case study and demonstrates the performance of the analysis methods presented in this work and their usefulness in the verification of a practical application. Chapter VII concludes with results of the research and future directions.

CHAPTER II

BACKGROUND AND RELATED APPROACHES

This chapter provides a description of the general approaches to formal system verification. The dominant approaches, theorem proving and model checking, are described along with an efficient approach to model checking and symbolic model checking. The next section describes the symbolic representations used in symbolic model checking. Ordered binary decision diagrams (OBDDs) are a symbolic representation that has had much success in verification of large hardware verification problems and has been incorporated as the internal representation of many commercial VLSI CAD tools in the last few years. Also presented are Multi-Terminal Binary Decision Diagrams (MTBDDs), a variation of OBDDs utilized in this dissertation, and a brief overview of other OBDD variants and a few of the popular OBDD implementations. Several research-quality model checking tools are available for both reactive and reactive real-time systems. A description of selected model-checking based verification tools is presented.

Formal Verification

Due to the rate at which integrated circuit size and density is increasing, the size of the systems being designed is also increasing at the same rapid pace. Verification technology must keep pace with systems being verified. In the past hardware designs were simply verified using ad-hoc methods: testing of the actual system or using simulation engines, exciting the circuit with all possible input combinations (or as many as were feasible to try within time/cost constraints). Figure 1 shows a VLSI design flow. Currently extensive simulation via hardware CAD tools is the predominant technique used to test and verify hardware designs. Extensive verification is performed after the Logic Synthesis and Optimization stage, although simulation can be performed at multiple levels in the design process. Early in the design process, the abstract register transfer language (RTL) [2] level design can be simulated and checked for errors. As the design progresses to the gate level and finally the transistor level, simulation must continue to try to find errors that may have occurred during the implementation and refinement stages.

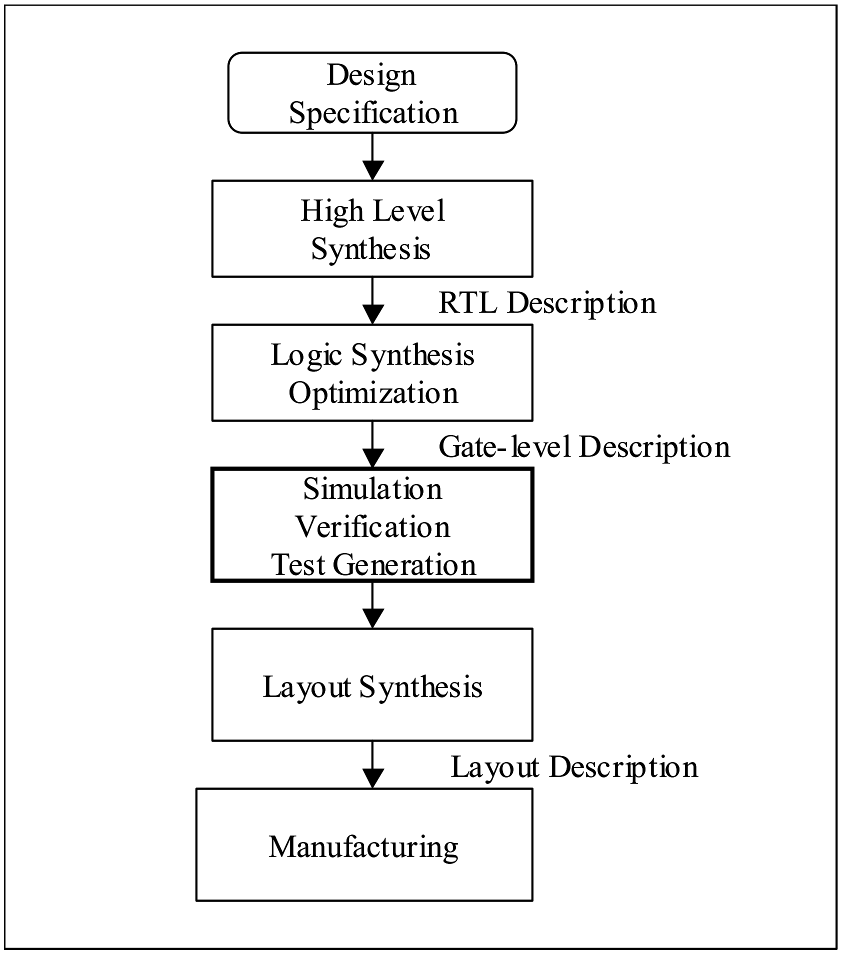


Figure 1: VLSI design flow

As designs are now well into the millions of transistors and with this number increasing rapidly, total coverage of the circuit behavior through simulation becomes a problem growing faster than the design problem itself. Exhaustive simulation is not feasible. The design may be broken into sub-systems for a more manageable size, but the interaction of these individual parts must be verified to operate correctly. When simulating circuits with a large number of possible states the probability of missing an error becomes more likely.

Software designs, high-level protocol descriptions, and real-time systems have typically had even less rigorous methods of testing. For verification of large-scale, system-level problems, more automated, efficient methods of verification are needed. Formal verification is attractive because it offers *complete* coverage of the entire operation of the system. In other words, formal verification is as good as exhaustive simulation.

Many different types of system design problems have been tackled using formal verification techniques:

- Circuit verification – RTL level, gate level, transistor level [6]
- Protocol verification – Communication protocols such as PCI, SCI [7]
- High-level system verification – Interactions between physical hardware, solid-state electronics, microprocessor operation [1]

There are two main types of formal verification being used today: theorem proving and model checking.

Theorem Proving

One type of formal verification is *theorem proving*. In theorem proving the system model and the system specifications to be proven are described in terms of mathematical statements. Verification proceeds by proving theorems about the system. The proof must show that the statement of the theorem can be formally derived from axioms using inference rules. Theorems must be developed and proven true in order to verify the system model satisfies the specifications.

Automatic theorem provers are available to assist in the mechanical details of this process but, in general, these are semi-automatic at best and have not made it into mainstream tools. The users must be experts of logic to perform the tedious task of writing the axioms to be proven [8]. The user must guide the verification process. Because of the level of knowledge required and the manual nature of theorem proving, this method of verification is an expensive process in terms of the time and training required.

Model Checking

Temporal logic model checking was first developed by Emerson and Clarke in 1981 [9]. In this approach, a finite-state transition *model* of the system is created that represents all possible behaviors. Then model is *checked* for satisfaction of a specification by exploring all possible behaviors with a model checking “engine”.

Specifications of the system to be verified are supplied in the form of a temporal logic. Model checkers exhaustively explore the state space of the state-transition system model to verify that the model satisfies the specification. Model checking provides coverage of the *complete* behavior of the system, not just behavior excited by a set of simulation vectors. An important point is that this procedure requires *no* user intervention, unlike theorem proving – the proving of the specification is completely automatic. If the model checker finds that the specification is not satisfied, most model checkers have the ability to return a counterexample. This counterexample is a state sequence that violates the specification.

The early model checkers used exhaustive analysis techniques and were limited to fairly small models by today's standards. The number of states that must be explicitly represented and checked using this method may grow exponentially in proportion to the number of sub-systems acting concurrently in the model [4]. When the models become too large for conventional model checkers to handle, model abstraction and state space reduction methods can be used to reduce the state space to a manageable size at the cost of some loss of information. This may be unacceptable for the verification of safety critical systems, for example. A fairly recent approach that has been developed in the last ten years is a *symbolic* method of model checking. Because this new form of model checking can allow a larger state space to be explored, the need to use reduction methods is lessened.

Symbolic Model Checking

The limiting factor of model checking is the representation of the extremely large state spaces that can occur. When the system has components operating concurrently that may transition in parallel, the state space grows as the product of these concurrent state spaces. Since the introduction of model checking, advances have been made to make the representation and manipulation of these large state spaces feasible. In 1987, McMillan realized that the state-transition model could be much more efficiently manipulated if represented implicitly with *ordered binary decision diagrams* (OBDDs) [9]. OBDDs were introduced in their current form by Bryant in 1986 [10]. This new form of analysis that operates directly on the model represented by symbolic equations rather than explicit enumeration of states is known as *symbolic model checking*. Although still limited by the state explosion problem, the OBDDs go a long way to mitigate the problem.

Symbolic model checking has proven to be successful because of the use of OBDDs to provide an efficient representation of the state space. The biggest gains of using a symbolic method are realized when analysis algorithms can operate completely within this symbolic domain. For example, reachability analysis can be performed to find the set of states that are reachable from a given initial state set. This algorithm operates only on state *sets* rather than individual states, providing a much more efficient method than a standard traversal of the state space that must examine individual paths.

Symbolic methods also work well when examining the behavior of non-deterministic systems. A traditional simulator examines a single state sequence path. Symbolically we can, through the use of state sets, examine all paths simultaneously. This is essential for the exhaustive analysis of large systems - systems too large to verify through the use of exhaustive simulation. A verifier must examine *all* possibilities.

Although oriented towards the verification of hardware circuits, the graph shown in Figure 2 gives an idea of the current state of design verification tools and the relative amounts of design coverage provided for given design sizes. On one end of the spectrum, model checking provides complete coverage but with a limit to the size of the systems it can cover. Simulation can operate on the largest of designs but has a very limited amount of design space it can cover.

The practical use of symbolic model checking is limited by the state explosion problem as was non-symbolic model checking but now the sheer number of reachable states is not necessarily the limiting factor in system verification. The size of the OBDD that represents the system is now the bottleneck. One goal of this work is to improve on the use of the OBDDs for the symbolic representation of complex systems modeled at a higher level of abstraction such as modeling the interactions system components, as opposed to the modeling of a hardware circuit. The more efficiently the model can be represented symbolically, the larger the models that can be analyzed.

A much more difficult problem is the verification of timed systems. These systems can exhibit exponential growth with respect to the number of clocks. An efficient method is also needed to represent these systems symbolically.

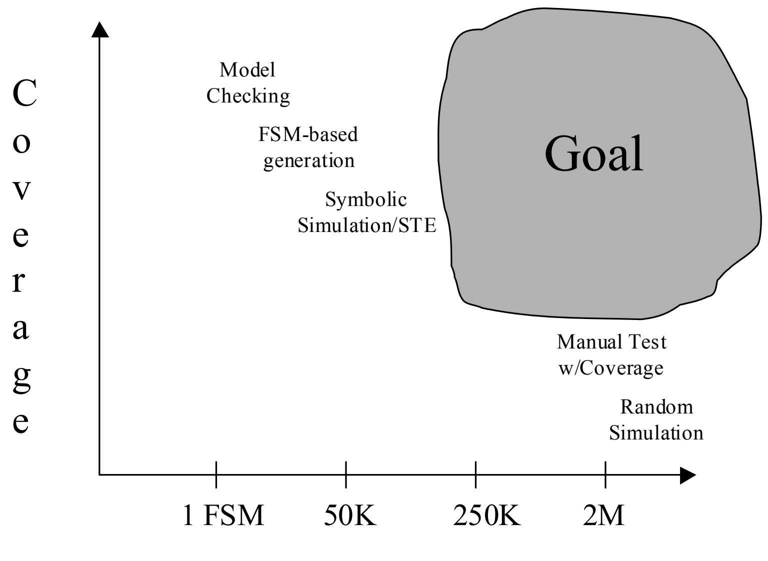


Figure 2: Coverage vs. Scale

Symbolic Representations

Ordered Binary Decision Diagrams are widely used with success in several areas including verification of VLSI designs and symbolic model checking. Also presented in this section are variants of OBDDs, one of which is the Multi-Terminal BDD (MTBDD) that is also used in this work.

Ordered Binary Decision Diagrams

Ordered binary decision diagrams, as proposed by Bryant, are a canonical structure used for representing Boolean functions [10]. The OBDD structure is usually a more compact representation than traditional sum-of-product/product-of-sums representations especially for large functions. Operations can be performed on OBDDs in a very efficient manner. These properties have in recent years made OBDDs popular for use in a variety of systems where very large state spaces must be managed efficiently.

OBDDs take the form of a directed acyclic graph. Each internal node of an OBDD is labeled with a variable of the function the OBDD is representing. For each internal node there are two edges leaving that node labeled 1 and 0. Every OBDD has exactly two leaf nodes that represent the values *true* and *false*. The graph structure described forms a type of decision tree where all variable assignments that satisfy the represented function *f* lead to the *true* leaf node and all other assignments lead to the *false* node. The OBDD structure has a single root node. As the name implies the variables have a strict fixed ordering. A consequence of this is, as a path is traversed from root to a leaf, or terminal node, each variable node is encountered only once. OBDDs are a reduced, canonical structure also referred to as ROBDDs or BDDs. Due to their canonical property, OBDDs representing the same function are isomorphic. This property allows efficient equivalence checking. Figure 3 illustrates the OBDD for the Boolean formula $(a \vee b) \wedge c$.

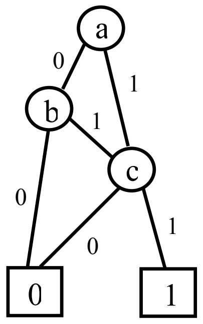


Figure 3: OBDD for $(a \vee b) \wedge c$

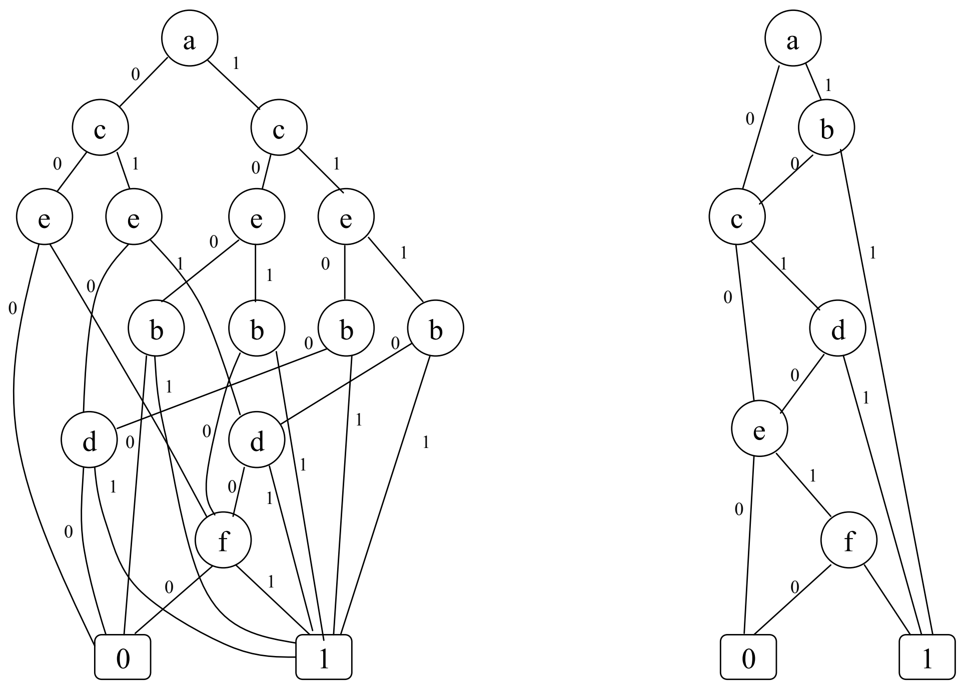
Basic OBDD functions include the standard Boolean operators: equivalence, AND, OR, NOT, etc. These algorithms have time and space complexities that are polynomial with respect to the number of OBDD nodes of the operands. This means that for reasonable sized OBDDs the worst-case performance of an operation remains reasonable [11]. OBDDs do have their drawbacks. There are some functions for which OBDDs simply cannot provide an efficient representation, such as a multiplier. An OBDD representation of a multiplier requires a number of graph nodes that grow exponentially with respect to the input word size [10]. With that said, OBDDs do perform well for the representation of many practical applications.

Variable ordering

The ordering of the variables (the ordering of the levels within the graph) can have a dramatic effect on the size of the OBDD. A bad variable ordering can greatly increase size and time required to perform operations on that OBDD and, sometimes more importantly, the OBDD may be too large to fit in the computer's physical memory. Using virtual memory is usually not an acceptable solution since memory accesses to OBDD structures are very irregular and causes a large amount of page faults.

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 may only be of linear growth [12]. By using common sense in choosing the variable ordering or using a heuristic developed for the particular application, these problems can usually be avoided.

The example in figure 3 shows two OBDDs with different variable orderings representing the same function ($f = (a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$). The OBDD on the right has a well-chosen variable ordering. In this case the variable ordering is optimal for this function. Only 6 nodes are needed. The OBDD on the left has a variable ordering that produces an OBDD that requires 14 nodes to represent the same function.



Ordering: a < c < e < b < d < f Ordering: a < b < c < d < e < f

Figure 4: OBDDs for f = $(a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$

Some applications using a static variable ordering throughout application may not produce acceptable results. In this case *dynamic variable ordering* may be needed to maintain manageable OBDDs. Dynamic variable reordering is a method for automatically changing the variable ordering to find an improvement in the size of the OBDDs. Many OBDD packages have support for dynamic variable ordering [18][19].

Many heuristics for dynamic variable ordering have been proposed. These heuristics can be divided into two categories: reordering on a global level and reordering on a local level. A common local reordering method is known as *sifting*. Sifting is a method to swap adjacent variables to search for a better variable ordering (fewer OBDD nodes). Reordering on a global level proposes a completely new variable ordering. This global rebuilding is much more time consuming but also has greater potential gains.

Dynamic variable reordering is an attractive option because it is fully automatic. The only drawback is the large amount of time it takes to search for a better variable ordering. Using this strategy can sometimes be the difference between completing an operation or it ending in failure because system memory has been exhausted, although a performance penalty is paid. It can slow down the application by as much as a factor of 10 [13].

MTBDDs

Many extensions to the basic OBDD form have been proposed. One of these extensions is the multi-terminal BDD (MTBDD). The MTBDD represents *pseudo* Boolean functions [14]. MTBDDs map bit vectors of Boolean values (which may represent a set) to a finite set of elements (integers) instead of only to the values true and false as in standard OBDDs.

More formally, MTBDDs are used for representing functions of the form $f: B^n \rightarrow N$, where *f* is a function that maps Boolean vectors of length *n* to integer values [15]. The function *f* effectively partitions the space of Boolean vectors, B^n, into *M* sets, where *M* is the number of integers the sets are being mapped onto. A similar representation to MTBDDs would be to use an array of OBDDs, with each OBDD representing a set that can be associated with a corresponding integer value.

MTBDDs range is Boolean, but their domain is arbitrary, defined to be the set of integers, N. Figure 5 below shows an MTBDD function, $f=4x_2+2x_1+x_0$ which maps 3-bit binary numbers into the corresponding natural number [16].

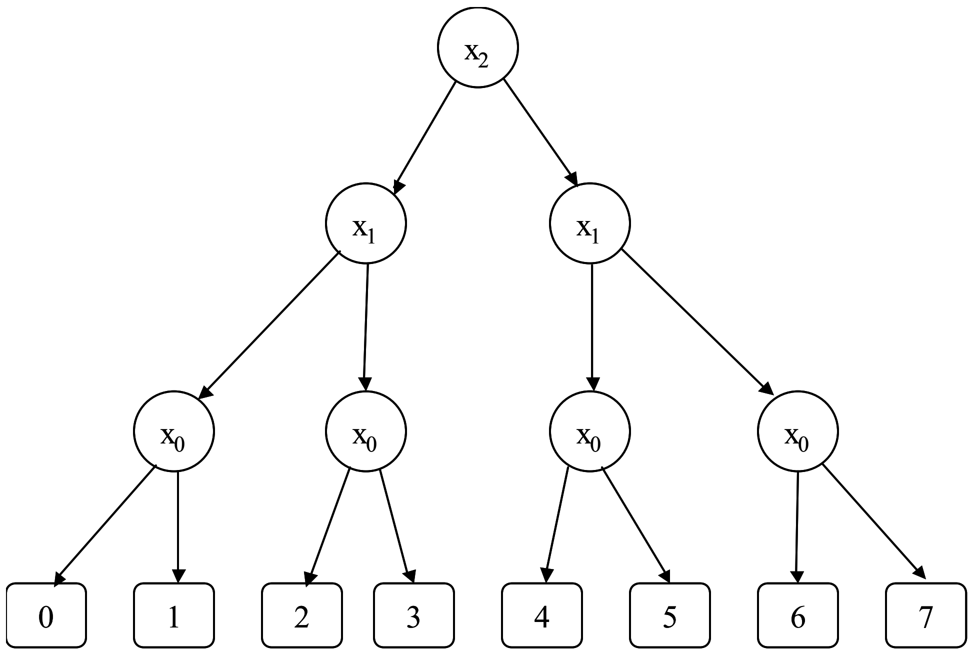


Figure 5: MTBDD mapping 3-bit binary number to its natural number

The reduction rules are the same for MTBDDs as they are for OBDDs and MTBDDs exhibit the same canonical property as OBDDs. However, since the image/result of function represented by MTBDDs is a member of N, Boolean operations are no longer applicable [17]. With that said, the mechanics of operations applied on MTBDDs work in much the same way as OBDDs. The *apply* routines traverse the MTBDDs beginning at the root applying a Boolean operation on the way down the structure. When terminal nodes are reached the terminal transformation is applied. This transformation takes two terminal nodes and produces a new one. In this way, same as OBDDs, new graphs are built up from the recursive application of a transformation function.

OBDD implementations

The efficiency of the techniques used to implement the OBDDs and the operations on them are critical to their performance. During the course of any OBDD operation many OBDD nodes are created and destroyed. For operations on large OBDDs hundreds of thousands of nodes may be created and destroyed in the intermediate steps of a single operation. Efficient memory management is crucial. Since all nodes are of the same size, structure overhead can be reduced by using a garbage collection strategy. In one such strategy, deleted nodes are not immediately *free*'ed but are put in a free list to be reused if possible so that expensive operating system *free/malloc* calls can be avoided.

Most OBDD packages have a type of operation cache where computed functions are stored. This cache stores the operation performed and its corresponding resultant OBDD. *Strong canonicity* should be always guaranteed by the OBDD implementation. Strong canonicity defines that in an OBDD implementation if f and g are equivalent functions, then not only are their representations the same, but they are, in fact, represented by the same structure in memory [16]. The pointer that points to the OBDD of f and g will point to the same structure. As a result of this property, to perform a test to determine if two OBDDs are equal, the user must only compare their pointers to determine equivalence. If they are equivalent then they must point to the same structure [5].

There are several OBDD packages that exist, each with different features. Some features that separate the implementations are efficiency of the package, the methods of dynamic variable ordering that the package provides, and support for other OBDD variations that may be needed. The most notable OBDD packages are:

- bddlib: *bddlib* was developed by David Long at CMU. This implementation has been widely distributed both stand alone and as a part of SIS system discussed later [18]. This package has support for dynamic variable ordering through the use of sifting and window permutations. *bddlib* has some limited support for MTBDDs.
- CUDD: *CUDD* was developed at the University of Colorado at Boulder by Fabio Somenzi. CUDD has support for dynamic variable ordering using random variable exchange, sifting, group shifting, window permutations, simulated annealing, genetic algorithm-based reordering, and identification and linking of symmetric variables. CUDD supports OBDDs, ADDs/MTBDDs, and ZBDDs. This package shows a good tradeoff between memory usage and execution time [20].

Both *bddlib* and *CUDD* packages are used in this dissertation work. *bddlib* is used for OBDD implement and *CUDD* is used to MTBDDs implementation.

Model Checking Tools

SMV

The SMV (Symbolic Model Verifier) system is a tool developed by McMillan at Carnegie Mellon University for checking modeled finite state systems against a specification described using a temporal logic (Computational Tree Logic, CTL) [9]. SMV was designed to be an experimental tool for exploring the practicality of model checking and how it applies to hardware verification. SMV has become somewhat of a point of reference in the model checking research community. Many researchers exploring state-space representation and traversal optimization techniques have used SMV as the platform for implementing their own algorithms and extensions [21].

The SMV language is used to describe a finite transition relation model. This relational model is represented symbolically as an OBDD. Properties of the model to be verified are specified in a temporal logic, CTL. Efficient OBDD-based algorithms are used to verify that the model satisfies the CTL specifications. If the model checker finds that a specification can be violated, a counterexample may be generated which demonstrates a sequence of events in the model that leads to a fault.

The language provides for the descriptions of reusable *modules* and hierarchical definitions. Synchronous and asynchronous models may be described. In a synchronous composition of modules, when a single step of this composition is taken, a single step is taken in each of the modules. In an asynchronous, or *interleaving*, composition of modules, when a step of the composition is taken, a step is taken by exactly one component [1]. The SMV language also provides for the description of non-deterministic behavior. Although the SMV language supports arithmetic integer operations, it has a very inefficient implementation. SMV also does not support a true timed model. If timing is to be represented, the underlying model will be a series of states with each state representing the passage of one unit of time. It is easy to see how this method is not feasible for systems with large delay times.

NuSMV is a reimplemented and reengineered of the original SMV model checker, introduced in [22]. NuSMV supports a user interface with an interactive shell. To support larger state spaces, NuSMV implements both the conjunctive and disjunctive partitioning methods of [23]. The modeling language remains the same along with the temporal logic specification language, CTL. Support for model checking of LTL specifications was added via a reduction of LTL specifications to CTL. Also, the underlying OBDD implementation was changed to use the CUDD package [19]. The current release is NuSMV 2, which adds model checking techniques based on prepositional satisfiability techniques (SAT) [24].

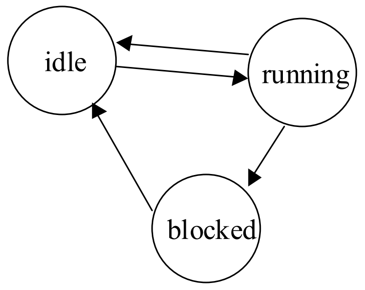
Computation Tree Logic, or CTL, is a propositional, temporal logic used in SMV to express properties of a model to be verified. The CTL specifications operate on a *computation tree* model of the system. A computation tree is a structure derived from a finite state graph by unwinding the state trajectories and producing a tree structure [25]. The computation tree represents essentially the traversal of a finite state graph beginning at a declared root state. The root may be any state; it is the point at which the CTL property is referenced. A CTL formula describes properties of the paths leaving the root state. CTL is classified as a *branching time* logic since it can describe properties of multiple paths leaving the reference state.

CTL formulas are composed of atomic propositions, Boolean connectives, and temporal operators. The Boolean connectives are $\neg$ (not), $\wedge$ (and), and $\vee$ (or). The temporal operator is composed of a path quantifier and a linear time operator.

The linear time operators used to express temporal relations are as follows:

X f	state f is immediate successor
G f	state f holds for all states on path
F f	state f exists in the future
f U g	from state f until state g

The linear time operators describe behavior along a single path. The path quantifiers are branching time operators meaning they operate on a branching tree structure. There are two path quantifiers, *A* and *E*. The universal path quantifier *A* represents that *all* paths are referenced from the root state. The existential path quantifier *E* represents that the property references *some* path from the root state. The possible CTL operators are enumerated as: EX, EG, EF, EU, AX, AG, AF, and AU.



Specification: idle → EX running

Figure 6: Sample state transition graph with CTL specification to be proven

A simple example of computing CTL constraints is the verification of the formula $S_0 \rightarrow EX\ a$. From Figure 6 we can choose *a* to be state *running* and S_0 to be state *idle*. The CTL formula EX *a* must be computed. EX *a* represents the set of all states that may lead to the state set *a* in a single step. From the system model EX *a* is the state set that results from a single backwards step from state *a*. For the example above, EX *running* is equal to the state set {*idle*}. In this case the specification *idle*→ EX *running* is satisfied since S_0 is contained in the resulting state set.

The combination of these two types of operators allows for the description of many temporal properties of systems that can be checked such as safety properties and freedom from deadlock. For example, specifications such as: *the brake caliper will always activate (sometime) after the break pedal is pressed* (AG (pedal_press->AF activate)), or *the brake caliper will always activate the next time (next system state) after the break pedal is pressed* (AG (pedal_press->AX activate)). CTL cannot express quantitative temporal properties such as: *the brake caliper will always activate within 1 second of a brake pedal press*.

Efficient algorithms exist for verifying that the system model satisfies the CTL specification [16]. These algorithms use the symbolic model of the system that is in the form of a Boolean next-state transition relation. This equation is, in turn, represented in the form of an OBDD. Using the OBDD representation of the system model there is no need to explicitly build the computational tree structure to verify the CTL specifications. A CTL formula can be identified by a set of states (represented with an OBDD) derived from the model that satisfies the formula. The model checking of the CTL formula only requires traversal of the model and manipulation of sets of states. For example, to compute EX*a* a single backwards step (sometimes called pre-image) is computed. Using the OBDD representation to verify the specifications provides an efficient method of model checking.

Although CTL specifications are able to assert many of the desired properties of systems, it cannot describe important quantitative information that is essential in verification of real-time systems (of course, this also assumes an underlying timed model structure would be needed). To address this problem several other variants of CTL have been proposed that include quantitative timing information to describe specifications for these types of systems. SMV does not support any of the timed variations of CTL.

Verus

Verus is a system used for the specification and verification of real-time systems. The goal is to provide insight into *how well* a system works rather than just determining *if* it works by way of computing its timing characteristics [25]. The Verus language is used to describe the system and its temporal characteristics [26]. CTL and RTCTL (real-time CTL) expressions may be used to specify properties of the system to be verified. RTCTL is an extension of CTL that places timing bounds on all CTL operators. For example, a common RTCTL temporal operator is the *bounded until* operator (derived from the CTL U or "until" operator). The bounded until operator has the form $U_{[a,b]}$, where an interval $[a, b]$ defines the time interval in which the property must hold true. The formula $f\ U_{[a,b]}\ g$ is true of some path if 1) g holds for some future state s on a path, 2) f holds for all states in between s_0 and s on this path, and 3) the distance from s_0 to s is within the specified interval $[a, b]$ [1].

This specification of the system is then translated into a state-transition graph. Model checking algorithms are used to verify the specified properties and extract performance data about the system. Schedulability of processes can be verified by finding maximum times between processes execution. Verus has the capability to give a counterexample sequence of states when a property is not satisfied. The Verus system can also be useful with untimed models as well. Campos claims models with up to 10^{30} states can be verified in minutes with this system [27].

The Verus language is a simple language that looks similar to C. Variables may be local or global (to a process). Concurrent processes may be modeled with interaction possible through global variables. Variables are also identified as internal or external. External variables are set from sources outside the system that are not influenced by the model. The value of external variables can change non-deterministically at any transition of the model. Variables may be defined as one of two data types: integer and Boolean.

Several primitives specific to temporal model definition are available:

wait - The wait operator is available to represent the passage of time. All other operations are defined to occur in zero time. The state of a process can only be observed when a wait condition occurs.

select - The select operator is used to explicitly define non-deterministic action in the program flow.

periodic - The periodic statement is used to declare a section of code enclosed in brackets as executing periodically. This statement has several parameters:

start_time - the time that will pass before the code begins execution for the first time

period - the period at which the code will execute

deadline - if the code does not finish in this length of time an exception will be raised

Deadlines may also be specified in aperiodic processes using the *deadline* statement which declares that a segment of code enclosed in brackets must finish execution within the amount of time specified in the deadline statement. When a deadline, either defined by the periodic or deadline statement, is missed, an exception handler is executed. The exception handler is a segment of code within brackets designated by the keyword *handler*. After the handler code is executed, the rest of the code in the deadline scope is ignored and control resumes at the next statement after the deadline code.

Verus has some command-line options that can enable features to increase performance. Normally, Verus uses a single monolithic transition relation to represent a system that may be a composition of the individual transition relations of concurrent processes. Computing the composition of the smaller transition relations (in terms of OBDDs) can cause a state explosion. An option is available to keep these individual transition relations separate. This option generally uses less memory and may be necessary for some systems. It does complicate the computation of a basic step since multiple relations must be dealt with.

Verus also has support for automatic variable reordering of the model before the global transition relation of the system is constructed. This reordering process permutes variables to produce a smaller OBDD representation. Since this can be a very lengthy process, support is provided to save a specific variable ordering to a text file that may later be reloaded instead of performing the reordering process again. Campos states that it is beneficial to experiment with the variable ordering as the model is being built. Since the automatic variable reordering moves single variables to produce incremental improvements, better results can be obtained using this method on small sections of the model rather than simply beginning with the final large model.

Manual reordering may be also used to find a more compact OBDD. Common sense approaches such as placing variables that make global decisions about the system should be placed first in the ordering. Variables that are closely related should be placed together (such as present state and next state variables). These are only general recommendations and may not apply to all types of systems.

The current released version of Verus is 0.9. Some of the special primitives of the language have not yet been implemented. This version only supports unit-time wait statements (wait(1)). To create a state that lasts 5 units of time 5 wait statements must be used. This makes the tool unsuitable for use on any system with large or widely varying times on transitions that cannot be reduced by a common factor manually at the modeling level. This tool simply expands the timed finite state model into a standard finite state model. The state transition times are presumably expanded to strings of unit length states. A state transition that is labeled 5 units of time is translated to a FSM form of a string of 5 standard states. This causes the internal representations of timed systems to grow at an exponential rate with respect to the number of states modeled. This current version does not produce counterexamples, but it will produce a restricted form of a counterexample by placing the EXAMPLE attribute on a CTL specification.

UPPAAL

UPPAAL is a set of tools for system validation and verification of real-time systems [28]. The UPPAAL system consists of a description language, simulator, and a model-checker. Systems that are suitable for verification with this tool include systems that can be represented with a collection of timed automata. These timed automata can represent non-deterministic processes with real-valued clocks. These processes interact through defined communication channels or shared variables.

UPPAAL allows entry of the timed automata description language in both graphical and textual formats (the graphical format is translated into the textual format). The simulation engine allows the user to interactively (graphically) examine the behavior of the model. Using the simulator, only a single execution trace can be followed. The simulator is provided as a means of debugging the model itself (or debugging of the system design if it is the beginning of a design process). The simulator also allows the user to graphically explore an execution trace that is produced from the model checker.

Extensions have been made to the UPPAAL description language to extend the timed automata with the use of data variables (real-valued integers and Boolean variables) to make the description language more like a high-level programming language. The possible values the variables may represent must be bounded or verification may not terminate.

UPPAAL attacks the state explosion problem through the use of symbolic modeling techniques and compositional techniques. The system is modeled as a parallel composition of automata in the form:

$A = (A_1|A_2|..A_n)$

The state space is explored in terms of symbolic states in the form of (*l*, *D*) where *l* is a control node vector of length n [28]. An element of *l, l*$_i$ corresponds to a state in A_i specifying the current state of each of the timed automata and *D* is a set of clock constraints. A symbolic state (*l*, *D*) represents a set of all the states (*l*, *u*) where the clock variables *u* satisfies the constraint *D* [29]. In this representation the data and shared variables are treated in much the same way as clock variables. A model-checking procedure is used in UPPAAL to determine whether or not the timed automaton A satisfies a given formula φ. The property φ must be of the form:

$\varphi := \forall\Box\beta \mid \exists\Diamond\beta \qquad \beta := \alpha \mid \beta1\wedge\beta2 \mid \neg\beta,$

where α is an atomic expression that may be a clock constraint or a state constraint. $\forall\Box\beta$ denotes the expression β must be satisfied for all reachable states (from a given initial state set). $\exists\Diamond\beta$ expresses β must be satisfied for some reachable state set [30].

From an algorithm point of view the model checking of the specification proceeds much as it does for other finite-state model checking reachability analysis algorithms. The state space is explored in a breadth-first manner until the real-time property is proven true or false. For the examination of the state space, UPPAAL also uses an "on the fly" verification method. The full state space of the automaton is never explicitly produced.

This method requires more overhead on each iteration but avoids the enumeration of the entire state space. If the verification is limited to a single automaton, then only that part of the model will be examined. Using the on-the-fly traversal method if similar properties are to be verified then work must be duplicated. In a recent version of UPPAAL steps have been taken to reuse portions of the computed reachable state space [31].

A large part of the work UPPAAL must do in traversing (and computing) the state space during verification is the repeated application of clock constraints. To aid in the computation of the set of successor states an efficient data structure known as *difference bounded matrices* (DBM) is used for the representation and manipulation of clock constraints [32]. This canonical structure holds all possible constraints for any clock or pair of clocks. When computing the successor states it is important to be able to quickly determine the intersection between constraint regions. This structure is also used in KRONOS.

The description language also supports the modeling of systems with a “simple linear hybrid automata.” This structure is a timed automata with clock values on the transitions that are not a fixed value, but are defined to be a clock interval with upper and lower bounds. This non-deterministic interval structure is then transformed into the standard timed automata supported by UPPAAL [28]. There is no specific support for clock intervals in the underlying symbolic representations used for model checking. The author does not make claims about the efficiency of the standard timed automata that results from this translation.

KRONOS

KRONOS is a tool for the verification of safety and liveness properties of real-time systems and the domain of timing analysis of hardware circuits. The system model is a set of concurrently operating timed automata with a finite set of real-valued clock variables. The values of the clock variables increase uniformly to denote the passage of time. KRONOS uses a timed extension of CTL, TCTL [33], as a means of formally describing the quantitative temporal properties of the timed-automaton model to be verified.

KRONOS uses symbolic representation methods very similar to those used in UPPAAL to attempt to control the state explosion problem. Instead of representing single states, a symbolic representation is used to represent a set of states. The set of states is represented by a system of linear constraints over the clocks of the timed automaton. These clock variables of the systems can take on any positive real value. A structure called a *region graph* is suited for representing finite portions of this infinite state space [34]. Sets of equivalent states are represented by nodes, or regions, of a region graph. Essentially, the sets of equivalent states, or regions, describes the unique *states* of the system for verification purposes. These regions can be manipulated to traverse the state space. The disadvantage to this approach is that the size of the region graph is exponential with respect to both the number of concurrently operation processes and also the number of clock variables present in the system. This region graph becomes quickly unmanageable. This approach is basically an enumeration of the state space.

To improve on this approach, KRONOS has implemented an on-the-fly approach to the exploration of the state space. In this approach, a symbolic graph called a *simulation graph* is constructed. This simulation graph is much smaller (because of its implicit symbolic representation of the state space) than the region graph. The simulation graph only explicitly models the transitions between discrete states. The passage of time occurs implicitly inside the state nodes [34]. The verification algorithms perform a depth-first search of this simulation graph that is constructed “on the fly”. A single stack is used to keep track of the current path of the simulation graph being explored. Using this method it is not required that the entire simulation graph be completely constructed in memory. Although the individual states of this graph represent a state set or region of the state space, a disadvantage to this method is that all simulation graph trajectories must be explored individually.

The cost of verification (size of state space) grows drastically as the number of system clocks increase. Another tool suite known as Optikron has been developed to reduce the number of clock variables of a timed automaton without changing its semantics [35]. Optikron can be used to reduce the complexity of the model before handing it off to the verification tool. Two main clock reduction techniques are used in Optikron. The first technique searches the model to detect *active* clocks – clocks that are necessary to produce the desired behavior of the system. If a clock is found to not be active then it can be safely removed without changing the system behavior. The second method works to detect clocks that are always the same for any location in the model. If a pair of clock variables is found to be the same value for all cases, then they can be collapsed into a single clock variable [36].

Summary

The tools Uppaal and Kronos both verify systems using a fairly low-level timed automata model. Neither method has the facilities for modeling *system* level concepts such as processors/processing elements, high-level model descriptions, etc. They are both for *design verification*. This work attempts to provide accurate *design and implementation verification*.

CHAPTER III

MODELS OF COMPUTATION

This chapter presents the models of computation used in this dissertation. A model of computation defines the interactions that may occur among components. In [37], Lee refers to a model of computation as *"laws of physics"* that govern component interactions. It is the programmer's model, or the conceptual framework within which larger designs are constructed by composing components.

The definitions of the existing FSMs, Statecharts, and Timed Automata are reviewed. A new statechart-like model, RScharts, is introduced. A timed extension of the RSchart model, named TRScharts, is also introduced. TRScharts extends the RSchart definition with the concept of states that have a finite, discrete duration. It was necessary to define these new models because the feature set and step semantics vary slightly from other defined statechart variations.

The dataflow model of computation is reviewed in a general sense. Note that the dataflow model of computation specific to this dissertation (Real-Time Dataflow) is introduced in Chapter V.

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. When a state change occurs, the next state is chosen based on the system inputs and available transitions.

A finite state machine is a 4-tuple, $R = \langle X, S, NS, S_0 \rangle$ where,

X is the input alphabet

S is the set of all states

NS: next state transition function, $S \times X \rightarrow 2^S$

S_0 is the initial state

FSMs are used not only for hardware description for synthesis but can be used for specifying behavioral descriptions of any finite state system. This work will exploit the fact that the FSM model can be used to model non-deterministic behavior. A current *system state* composed of both a state and input label from the set $S \times X$ can map into two or more states allowing the possibility for multiple transitions to be enabled simultaneously from a single state. This allows the model to represent multiple behaviors from a single state and given input conditions. Both deterministic and non-deterministic FSMs can be represented in the form of a single Boolean function (the *next-state* transition relation). FSM visualizations used for system modeling are sometimes referred to as *state transitions graphs*.

Statecharts

Large systems can require thousands of states to accurately model their operation. Using standard FSM models, these systems quickly become too cumbersome to comprehend. Most large systems naturally decompose themselves into concurrent behaviors. If these systems are modeled with a standard finite state machine (FSM), the number of states needed to represent the system behavior quickly explodes (known as the *state explosion* problem). Many languages have been developed to extend the basic FSM model to address this concern.

The most notable (and first developed) of these languages is the Statechart language [38]. Statecharts were developed by Harel for the specification of reactive systems (dominated by control logic) for use in the STATEMATE system, a commercial tool developed by I-Logix [39]. The Statechart language is a graphical language that extends the features of the traditional standard finite state machine with the concepts of hierarchy, concurrency, synchronous broadcast, and many other features.

The parallel composition of two state machines F and G will yield a system state space with the Cartesian product of these two state machines ($|G| \cdot |H|$ total states). Providing support in the language to represent hierarchy and concurrency explicitly allows us to represent a complex state space in a compact manner to avoid the state explosion problems (at least in the representation) that can arise in normal FSMs. This is important since many systems have inherent concurrency in their operation that would be very difficult to capture, and certainly difficult to understand, using standard FSMs. Use of hierarchy promotes top-down design practices and allows for modeling of systems at varying levels of granularity (although statecharts have additional features such as inter-level transitions that somewhat work against these design principals).

Each state can be composed of several sub-states that either operate independently or sequentially. Using Statecharts to model a system can greatly reduce the number of states needed to graphically represent system behavior and make the model much easier to comprehend. Consider a Statechart model as shown in Figure 7. This statechart has four independently operating parallel state machines (S_2, S_3, S_4, S_5) each having three sub-states. This model has 18 total states, whereas an FSM model would require 81 individual states to express the same behavior. Note that this more compact representation represents the equivalent behavior of the potentially large flat FSM model.

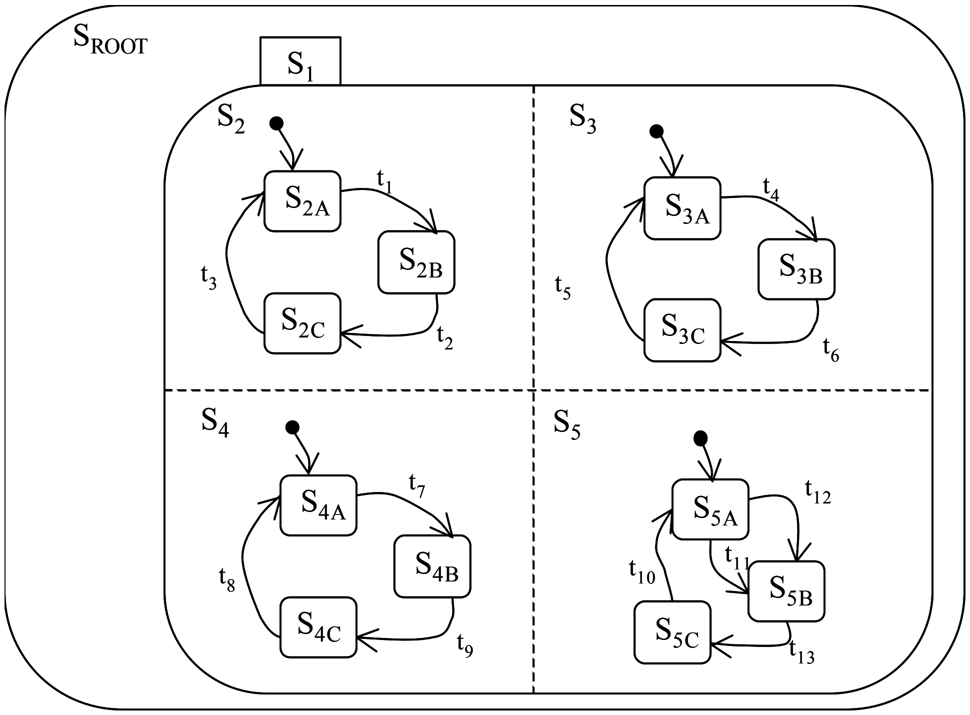


Figure 7: Example statechart

Defining the semantics of statecharts boils down to defining the step semantics – exactly how the system transitions from one state to the next and the actions that may take place around this state change. In the original Statecharts language as defined in [38] by Harel, and used in STATEMATE, implements two types of step semantics. The first is an asynchronous time model where the system reacts (takes a step) whenever an external change occurs and repeatedly executes a step until the system reaches a steady state called a *stable configuration*. This series of steps is known as a superstep and appears as instantaneous operation to the user. Each superstep takes one unit of time. In this case the system semantics satisfy the *synchrony hypothesis*. The synchrony hypothesis states that the system is always faster than its environment [39]. The system reacts to each input individually and the response to the input is seen as instantaneous. The second is a synchronous time model where at each clock tick the system responds to all external changes that have occurred since the completion of the previous step. The system only changes state at each clock tick.

Many variations of statecharts exist including a recent adoption of a statechart standard by UML [40]. Von der Beeck provided a good comparison of many (over 20) statechart variants in [41].

RScharts

The variant of statechart used in this work known as Restricted Statecharts ("RScharts") is described below. The RScharts model features are very close to the basic Statechart features as defined by Harel in [38]. Some of the original Statechart features not supported are: history mechanisms and the use of global variables. The syntax of RScharts is specified below.

Syntax

Formally, an RSchart is defined as ST = (S, T, E, S_0, Type, H) where,

S is a finite set of states, where each state, s ($s \subseteq S$), is declared as one of the three state types: {AND, OR, BASIC}

E is a finite set of events ($e \subseteq E$)

S_C is a set of states that forms a valid state configuration.

$T \subseteq (S_C \times 2^E \rightarrow S_C \times 2^E)$ is the finite, global state transition relation

S_0 is a set of initial states ($S_0 \subseteq S$) S_0 forms a valid initial state configuration.

Type: $S \rightarrow \{AND, OR, BASIC\}$ The state type mapping function.

H: S$\rightarrow$ S   The state hierarchy function. If s' $\subseteq$ H(s), then s' is an immediate descendant of s. The function H describes a state tree hierarchy of the model.

Root is the state at the uppermost level in the state hierarchy (Root $\subseteq$ S), Type(Root) = OR.

Each of the three states types of an RSchart is described below:

AND States. States of type AND enable modeling of concurrency by composing several simultaneously active sub-RScharts. AND states are parent states to sub-RSchart state machines that are concurrently active. These sub-RScharts may interact with each other via guarding conditions and events generated by other active components of the statechart. Immediate descendants of AND states must always be OR states. For a given s, if Type(s) = AND, then $\forall$ s'$\in$H(s), Type (s') = OR.

OR States. States of type OR support the embedding of one RSchart inside another to provide hierarchy in the model. OR states have sub-states that are related to each other by an exclusive-or relationship. Only one state may be active at a time. Both AND states and OR states are referred to as "superstates".

BASIC States. States of type BASIC have no sub-states. For a given s, if Type(s)=BASIC, then H(s)=$\emptyset$. The leaf states of an RSchart hierarchy must always be basic states. That is, if H(s)=$\emptyset$, then Type(s) must be BASIC.

An RScharts model, at any instant, may have multiple active states (if concurrency is modeled by the use of one or more AND states). This aggregate current system "state" of an RSchart is known as a *state configuration*. A state configuration always contains the Root state of the statechart. A state configuration must contain exactly one sub-state for each OR state and all sub-states for each AND state. A valid state configuration for the statechart in Figure 8 is: { Root, A, A_1, A_2, X_2, Y_3 }. Figure 9 is a tree structure depicting the hierarchical structure of the statechart in Figure 8.

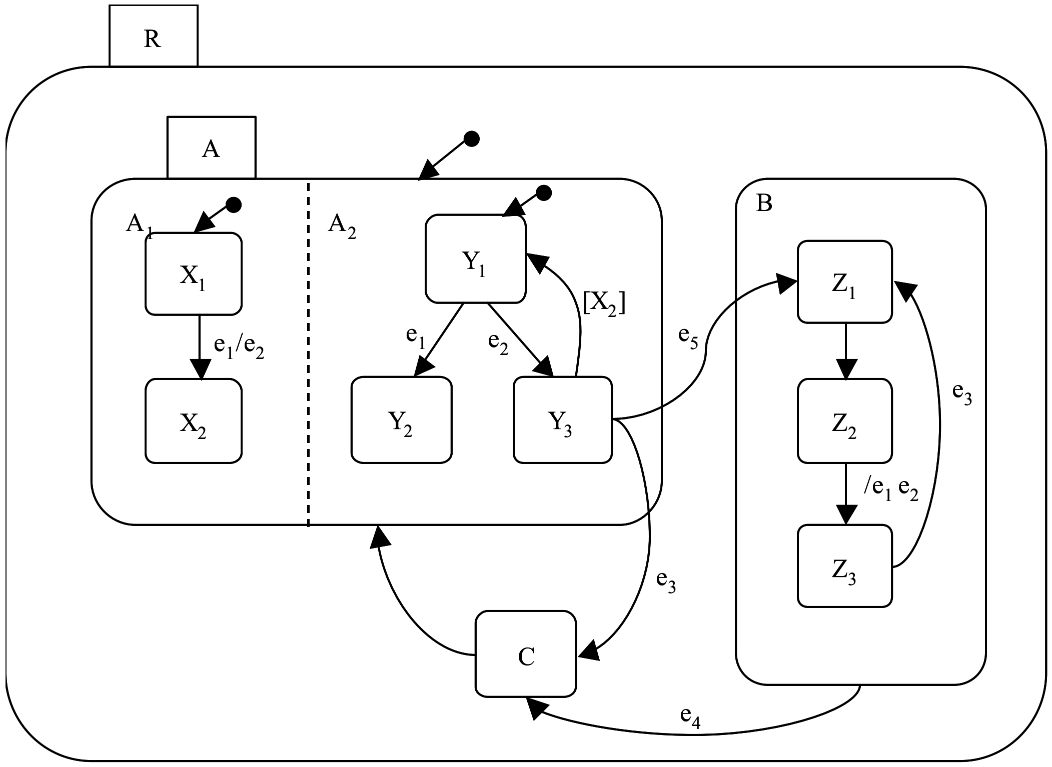


Figure 8: Example RSchart

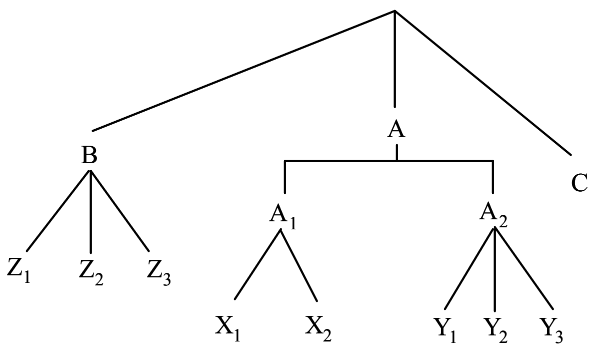


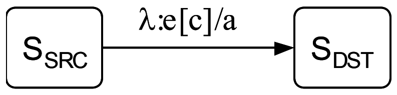
Figure 9: Example RSchart hierarchy structure graph of Figure 8

Harel states that state configurations are “closed upwards”. When a system is in any state, *s*, the configuration must also be in the parent state of *s* [39]. Note that, in reality, the set of BASIC states of a configuration are all that is needed to identify a state configuration. In Figure 8, for example, the set $\{X_1, Y_1\}$ is sufficient to represent the initial state configuration. This minimal set will be referred to as a *basic configuration*. The current overall system “state” of the RSchart is composed of the currently active state configuration along with the set of events that are currently active. In this paper, we will refer to this global system “state” of the statechart as a *global configuration*. In Figure 8 a possible global configuration of the statechart is $\{R, B, Z_3, e_1, e_2\}$.

Initial states are shown graphically as a sourceless arrow pointing to a state to be identified as an initial state. Each superstate must have an initial state in its immediate sub-hierarchy identified. The Root state is, by default, an initial state. The RSchart in Figure 8 has an initial state configuration of $S_0 = \{R, A, A_1, A_2, X_1, Y_1\}$.

The syntax of statechart transition labels (also referred to as *firing conditions*) as defined by Harel is *e[c]/a*, where *e* is an event expression that triggers the transition, *c* is a guard condition that prohibits the transition from being enabled if it is not satisfied, and *a* is the action that is performed if the transition is taken [38]. For our modeling purposes we refer to *e* as the *trigger expression*. The trigger expression is a Boolean expression composed of events and the Boolean operators *and*, *or*, and *not*. Events have global scope, i.e., all events are visible to all system components. The event variables represent events that may be produced internally or be presented to our system from an external source (internal events are generated by actions of transitions, external events represent events that happen outside the modeled domain). We refer to the guarding condition *c* as the *guard expression*. The guard expression is a Boolean expression composed of states used to express behavior dependant on the active states of other orthogonal components of the statechart. The *action expression*, *a*, enumerates the set of events that this transition will produce in the next system state.

Transitions labels are shown graphically in the form:



Where, given a transition $t \in T$,

S_{SRC} = source(t) is the source state

S_{DST} = dest(t) is the target state

λ: Transition label: e[c]/a, each of these (e, c, a) are optional.

e = trigger(t) – trigger expression – Boolean expression operating on events $e \subseteq E$. Expression may include logical Boolean AND, OR, and NOT operators.

c = guard(t) – guarding expression – Boolean expression operating on states $s \subseteq S$, where a and valid guarding condition has source(t) as a concurrent state (that is, a state with an AND state in its parent hierarchy) and its operands must be states orthogonal to source(t). Expression may include logical Boolean AND, OR, and NOT operators.

a = action(t) – action expression – defines set of events, $e \subseteq E$, that will become active for the next step.

Step Semantics

The step semantics implemented in the RSchart statechart variant is a synchronous semantics. In RScharts, active concurrent state machines progress simultaneously. If the current system state configuration contains AND states then the sub-RScharts of the AND states will all “take a step” and transition at the same instant. A transition will always occur at each step in each active concurrent space. If no explicitly modeled transitions are enabled, then an *implicit transition* will be fired. Implicit transitions are created to represent the fact that for a given (BASIC) state s, if we are in state s and no transitions are enabled then the system is to remain in state s. The use of implicit transitions ensures that at each system step, a transition will fire in *all* active concurrent spaces whether it is a modeled transition or an implicit transition.

An RSchart model ultimately progresses from one global configuration to the next. The synchronous semantics of the RSchart model allows it to be “flattened” into a single FSM preserving the model semantics. Each state of the flattened FSM represents a unique global configuration with all RSchart features removed. Ultimately, a system step, which may consist of multiple transition firings, can be viewed as a single global transition from one global configuration to another. The model checking algorithms in this work take advantage of this property and represent the entire RSchart in terms of a single transition relation model. The RSchart model supports the explicit modeling of non-deterministic behavior just as with the FSM model. In the case that we want to model a deterministic system, it is easy to inadvertently model non-deterministic behavior by mistake. Presented later is a method for testing for deterministic operation.

Concurrently active sub-RScharts interact through two mechanisms: events and guarding conditions. An event produced by an action in one sub-state machine may trigger a transition to occur in a corresponding concurrent state machine in the following time step.

Events are visible to all parts of the statechart hierarchy in a synchronous broadcast fashion. Following from the synchronous step semantics, event communication between the concurrent FSM spaces is also performed synchronously. Events persist only for the duration of one step. The explicit declaration that an event will not occur as part of the action expression of the firing condition is not allowed. If it were allowed, conflicts with transition actions in orthogonal states could arise. Multiple transitions may produce the same event; in this case the event is simply present as part of the global configuration produced as a result of the transitions fired for that step.

Guarding conditions are used to conditionally enable a transition from a state based on the state of another active concurrent sub-RSchart. Guarding conditions are only useful when the system has active concurrent sub-RScharts.

Inter-level transitions, transitions that cross state hierarchy boundaries as used in statecharts, are allowed. One type of inter-level transition originates from a superstate. These transitions behave in an interrupt-like, pre-emptive manner. If a transition leaving a superstate, *s*, is enabled, then the firing of all transitions contained within the sub-hierarchy of *s* is suppressed. This gives the transition leaving superstate *s* priority and preserves the intended deterministic operation. Inter-level transitions may also originate from within an orthogonal state and transition to a state outside of its orthogonal region. In this case the single transition actually represents a set of transitions. In Figure 8, the transition from Y_3 to C is enabled if event e_3 is present regardless of the state occupied in A_1. In the global scope, this is equivalent to transitions from X_1Y_3 to C and X_2Y_3 to C. Note that in general inter-level transitions work against the principle of model modularity and make it more difficult to decompose a large model into meaningful subsystems, especially in terms of a graphical representation.

Timed Automata

Timed Automata is a formal notation introduced by Alur, Dill [42] for modeling the behavior of real-time systems. A timed automata is basic automaton structure that has been extended to accept timed word sequences. In timed automata traditional state transition graphs are augmented with timing constraints. This model is classified as a *dense-time* model because the timing constraints operate on a set of real-valued clock variables. There is no restriction on the number of clock variables that may be used. The graph is composed of a set of states known as *locations* and a set of transitions between these locations known as *switches*. Time is elapsed in the locations; switches are instantaneous. A location may have an invariant (clock constraint) associated with it. If an invariant is present, then time may elapse in that state only as long as the invariant holds true. When the invariant becomes false, the automaton must transition to another state. If no invariant is present then the system may stay in that state and time may elapse for an arbitrary amount of time. Each switch may be conditionalized with an event. A clock constraint may also be associated with each switch. The switch may only be taken if the clock constraint and event condition is satisfied. Also, the switches can have a clock reset expression that resets a clock variable to a zero value upon firing. Essentially, the clock variables serve as individual timers, all progressing at the same rate and may be individually reset.

A timed automaton defines an infinite state transition graph with each state Q representing a pair (s, v) where s is a location and v is a clock assignment (a non-negative real value). Timed automata are compositional. A product construction of multiple interacting timed automatons is defined with an interleaving asynchronous semantics [1].

Clock variables are global and may be shared between automata to permit synchronization.

Timed Restricted Statechart (TRScharts)

For many reactive systems, standard untimed finite-state models provide a sufficient representation of system. For other systems, such as real-time systems, it is essential to include in the model not just the propagation from state to state, but also temporal aspects of the system. The addition of timing information is necessary to compute quantitative system performance characteristics and verify that system timing requirements are always met.

Timed restricted statecharts (TRScharts) are an extension of the RScharts model that provides us with a means to model the behavior of timed systems. Just as with RScharts, a TRSchart model is a synchronous model with the notion of a global step, where all orthogonal elements progress simultaneously. The system always progresses one unit of time at each step. Informally, each state is now required to be active for at least a duration, $t_{duration}$, with the duration $t_{duration}$ selected non-deterministically from a discrete time interval [t_{lower}, t_{upper}]. The selected time $t_{duration}$ is a discrete value that represents the mandatory time the system will remain in state s once that state is entered. Transitions leaving a state cannot be enabled to fire until $t_{duration}$ units of time have elapsed. This non-deterministic semantics allows us to model *all* possible cases where a state s must persist for $t_{duration}$ time units, specified by an interval.

Obviously, the choice of model used greatly affects ability to verify properties of the model. Some models are naturally more amenable than others to model checking methods. This model was selected because it retains most of the desirable properties of our untimed model in terms of its usefulness for formal verification.

A TRSchart is shown in Figure 10. Although the timing semantics of TRScharts may be somewhat inconvenient for a system description / input language, this model serves primarily as an intermediate representation of system behavior.

Formally, a TRSchart is defined as ST = (S, T, E, S_0, T, H, I, D) where,

(S, T, E, S_0, T, H) definitions are inherited from the RSchart model

I is a set of intervals [t_{lower}, t_{upper}], with $t_{lower} \subseteq N$, $t_{upper} \subseteq N$ and $t_{lower} \leq t_{upper}$, where N is the set of positive integers

D: $S_{BASIC} \rightarrow I$ is a duration function assigning a time interval, I, to each basic state.

TRScharts allow for a time interval [t_{lower}, t_{upper}] to denote a range of times that a BASIC state may persist. Superstates are used only as hierarchy and composition constructs and it is not necessary to denote the progressing of time in these states. For a hierarchical model, it is only necessary that time information be associated with BASIC configurations. The minimum duration of state is identified by an interval $t_{duration}$ = [t_{lower}, t_{upper}], where $t_{duration}$ may be chosen to be any integer where $t_{lower} \leq t_{duration} \leq t_{upper}$. After the system has occupied a state for $t_{duration}$ units of time, it may then transition to another state. Each possible $t_{duration}$ value (each value in the [t_{lower}, t_{upper}] range) creates a new behavior trajectory of the system. For wide variations between the upper and lower bounds of these interval specifications the state space grows dramatically.

For this timed model, the "state" of a non-concurrent model becomes a pair $\langle s, t \rangle$, referred to as a *timed state*, where t is valid $0 \leq t \leq t_{duration}$. For example, Figure 10 shows a TRSchart containing a state labeled "B" that is assigned a time interval [3,5] (t_{lower}=3, t_{upper}=5). A transition from state C will take us to the timed state $\langle B, 1 \rangle$. The mandatory time we must stay in state B is a non-deterministic value selected from the range [t_{lower},

t_{upper}]. In this example, the system may exhibit behaviors where state B may have one of three mandatory time durations: $t_{duration}$=3, $t_{duration}$=4, or $t_{duration}$=5. The possible behaviors the sub-TRSchart contained in the OR state S_1 may take from state C is:

$\langle B, 1\rangle \rightarrow \langle B, 2\rangle \rightarrow \langle B, 3\rangle \rightarrow \langle B, 4\rangle \rightarrow \langle B, 5\rangle \rightarrow \langle A, 1\rangle \ldots$

$\langle B, 1\rangle \rightarrow \langle B, 2\rangle \rightarrow \langle B, 3\rangle \rightarrow \langle B, 4\rangle \rightarrow \langle A, 1\rangle \ldots$

$\langle B, 1\rangle \rightarrow \langle B, 2\rangle \rightarrow \langle B, 3\rangle \rightarrow \langle A, 1\rangle \ldots$

In this example when $t_{duration}$ elapses and the system is allowed to transition from state B the transition from state B to state A will be enabled to fire since it has no triggering conditions and does not have a guarding expression. Of course, in general the system will remain in a state until an outgoing transition is enabled.

Each BASIC state present in a global configuration must have a time associated with it. The global configuration may contain concurrently operating state machines. The timed state of each of these orthogonal components must be captured by the pair $\langle s, t\rangle$. Note that for TRScharts the term *global configuration* is also used to refer to the global state of the system (just as with RScharts), but now the basic states within this set are paired with a time value. Event semantics have not been changed from that of RScharts. Events are active for the duration of one step.

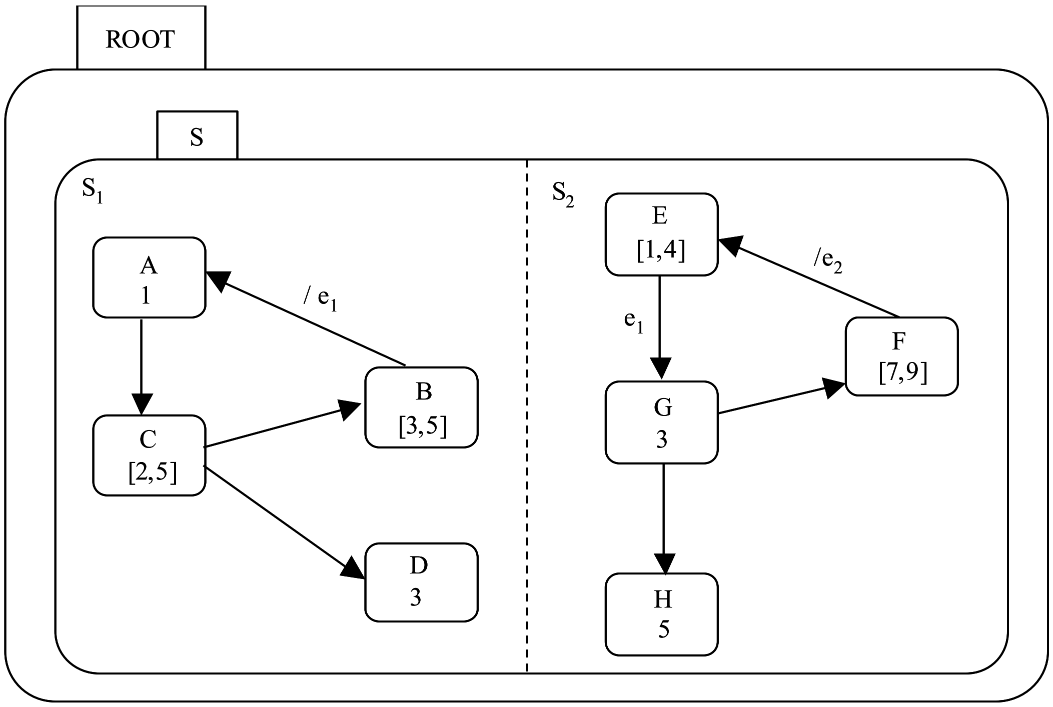


Figure 10: Example TRSchart

Dataflow Models

Data flow diagrams (DFDs) are a popular, easy to understand graphical notation. DFDs are very intuitive in that they fit the way designers naturally think about the signal processing computation flow. DFDs can serve as the bridge between the signal processing algorithm designer and the embedded system developer. Many CASE (computer-aided software engineering) tools are available that use dataflow program descriptions as graphical input design capture language (Khorus – University of New Mexico, SPW – Cadence, and COSSAP – Synopsis to name a few).

Dataflow Process Networks

Dataflow process networks are described by dataflow graphs, where nodes represent actors and the arcs represent communication between actors [43]. Tokens are the unit of data on which dataflow process networks operate. A token may represent a scalar, array, matrix, image, etc. Each arc represents a buffered communication pipe with a (possibly) unbounded FIFO (First-In, First-Out) queue. The system may operate on infinite streams of data. The actors perform some computation on its input data and output some resulting data. Each dataflow actor has a firing rule to determine when that actor is activated based on the tokens present on its inputs. When an actor fires, it consumes tokens on its input ports and produces tokens on its output ports. The data dependencies between actors (given by the dataflow graph itself) and the firing conditions of the actors define a partial ordering of an execution of the dataflow graph.

Synchronous Dataflow

Synchronous dataflow (SDF) is a restricted version of dataflow with several desirable properties [44]. Each node represents a computation (called an actor), and each dataflow arc represents a FIFO buffer transferring the data from one actor to another. A number may be present on the arc identifying the number of initial tokens in this queue. An integer label is present on each dataflow arc at the input and output of an actor node. For an input, this number specifies the number of tokens that must be present on that input port to fire the actor. For an output, the number represents the number of tokens produced on this output when an actor is fired.

Formally, an SDF model is composed of a set of actor nodes $n \in N$, and a set of dataflow arcs $d \in D$, where each arc, d_p, is a queue of data from one actor node, n_i, to another node, n_j. [45]. Each arc, d, has the following attributes:

A_P – number of tokens initially present on d_p.

W_P – number of tokens that must be present on d_p before n can be fired. W_P is also the number of tokens consumed by n_j (arc destination actor) from this arc.

U_P – number of tokens produced by n_i (arc source node).

Unlike the dataflow process network model, the SDF model implementation can be determined to have bounded queue buffers and guarantee freedom from deadlock. The key restriction of SDF is that each actor always consumes the same number of tokens on each input port and produces the same number of tokens on each output port. Fixed production and consumption rates lead to the ability to statically schedule the entire computation graph. A set of balance equations can be constructed from a minimal set of firings for each actor can be found, if it exists. If a solution does not exist, then a bounded memory solution to the SDF model is not possible and therefore an infinite execution of the computation graph is also not possible. A computationally efficient algorithm also exists to determine if a SDF graph can deadlock [44].

In addition to the desirable properties of guaranteeing bounded buffers and freedom of deadlock at the time of system synthesis, these properties also lead to an efficient implementation of a SDF models. Since the ordering of the invocation of the processes is fixed in the form of a static schedule no scheduling overhead is incurred and buffer sizes are only as large as necessary. The actors (processes) may be simply triggered directly at the appropriate time. Only a minimal or even no scheduler is needed, resulting in predictable execution. Also, since the maximum size of all token queues is known at compile time, no dynamic memory allocation is needed.

Summary

This chapter describes the basic finite-state and dataflow models of computation relevant to this dissertation. A new statecharts variant, RScharts, was introduced. RScharts have been successfully used as an input model for the behavioral description of physical systems as part of a safety and reliability analysis toolset. A timed extension to RScharts, TRScharts, was also presented. Both RScharts and TRScharts are used in this dissertation as an intermediate model to capture system behavior generated from an implementation of a more abstract system model.

The Dataflow model has proven itself to be a natural system design model for embedded signal processing applications. Two widely used dataflow models, dataflow process networks and synchronous dataflow, were presented as well.

CHAPTER IV

SYMBOLIC METHODS FOR REPRESENTATION AND ANALYSIS

Model checking analysis of large, complex systems is a challenging problem. Exhaustive analysis is needed to prove desired properties of the system such as freedom from deadlock, bounded memory usage, etc. are satisfied. Systems, especially embedded systems, often have a high-level of inherent concurrent and interrelated behavior. The state space of the system grows on the order of the product space of these concurrent concurrently operating sub-systems. The number of behaviors that must be examined depends on the amount of non-deterministic behavior present in the model. For the problem area focused on by this work, there is a need to model the following types of non-deterministic behavior:

1) In a purely token-based modeling paradigm only generic data token flow is modeled; the actual value of the tokens is not considered. System components may have data-dependent behavior. Although the actual behavior a component exhibits may not be known, we would like to model the range of behaviors that component may exhibit. For example, computation execution times may vary according to input data. Although the internal behavior of the computation has been abstracted away, we can model the task execution time as an upper and lower bound range. Also, a computation may have varying behavior based on its input data. In terms of a dataflow graph, the computation may or may not output a token based on some data dependency.

2) Behavior that varies due to some system characteristics not modeled. For example, computation runtimes may vary due to the state of the processor cache at the time a computation is executed. If data the computation needs to operate is not already present in the cache, then extra time is consumed to load the data.
3) Non-determinism in the system environment – e.g. jitter in periodic input.

The addition of timing information to concurrent systems adds another dimension to the representation of system behavior and causes an exponential increase in the size of the system state space. The exhaustive exploration of all the possible system behaviors is necessary to verify system properties. For standard methods of analysis, this exploration will take time proportional to the number of system states that must be examined. This limits the size and complexity of the systems that can be verified.

As described in Chapter II, symbolic representation methods avoid the explicit enumeration of these spaces. Symbolic model checking (SMC) has shown promise in the verification of very large state spaces [4]. This chapter will focus on the symbolic representation of statechart-like models. Symbolic representations are derived for both the RSchart model (similar to statecharts) and TRSchart model, a model that supports the modeling of quantitative temporal behavior of finite-state transition systems with a discrete model of time. Functions that operate on the symbolic representation of the system are presented that provide exploration of the state space. Reachability analysis algorithms are presented that are built on top of the basic symbolic traversal functions. The reachability algorithms can be implemented efficiently with symbolic methods if all operations are conducted on *sets* of system states. The symbolic analysis methods presented in this chapter provide a means of verification of large state space, non-deterministic systems.

Finite State Systems Symbolic Representation

OBDDs work well for representing large sets. An example of how an OBDD function can be used to represent a set is shown below. All elements must be assigned an encoding. A function can be created to define set membership (a.k.a. “characteristic function”) [11].

Example:
$$f(s) = \begin{cases} 1 \text{ if } & s \in S \\ 0 \text{ if } & s \notin S \end{cases} \tag{1}$$

For simplicity, in this paper any reference to an OBDD function representing a set refers to a set membership function representing this set. There may be instances where no distinction is made between the set membership function and the set itself. The OBDD representation of the set membership function has the interesting property that there is no relation between the number of elements in a set and the size of the OBDD representation of this set [46]. It is possible that very large sets can be represented by very small OBDDs and it is also the case some sets, *S*, will require an OBDD representation with size exponential with respect to |*S*|. In practice, however, the use of OBDDs can frequently give a manageable representation of very large sets.

We would like to develop a function that describes all behaviors of a standard finite-state machine (FSM) model. Let X = $\{x_0, \ldots, x_{n-1}\}$ be the set of variables used to represent a state and X’ = $\{x'_0, \ldots, x'_{n-1}\}$ be another set of state variables. A transition relation, TR(X, X’), is created that consists of terms identifying mappings from one system state to another. The transition relation, TR(X, X’) is created as the union of all transitions present in the FSM.

$$TR(X, X') = \bigcup_{t \in T} t(X, X'), \text{ where T is the set all transitions of an FSM} \tag{2}$$

Each state and event receives an encoding. Each transition expression, t(X, X'), is a relationship from a source state and its triggering event(s) (both expressed in terms of x variables) to a destination state and associated output event(s) (both expressed in terms of x' "next-state" variables).

$$\mathrm{TR}(X, X') : \ B^n \times B^n \rightarrow B \qquad (3)$$

where n is the length of the encoding vector

The symbolic computation of finding a next state set from a current state set is known as finding the *Image Computation*. Let S(x) be a membership function that represents a set of Boolean-encoded *system state(s)*. An Image Computation function is created to find a next-state set from a current-state set.

$$\mathrm{Image}(\mathrm{S}(X)) = \left(\exists_X \left(\mathrm{S}(X) \wedge \mathrm{TR}(X, X')\right)\right)\Big|_{X' \rightarrow X} \qquad (4)$$

The operation, S(X)∧TR(X, X'), returns the state transition mappings from the system states in the current system state set, S(X), to their respective next system states (in terms of X'). We would like the Image function to ultimately return the next-state set in terms of X. To do this we first existentially quantify out the state vector x variables from the selected state transition mappings (result of S(X)∧TR(X, X')). We now have a function representing only the next-state set, but represented in terms of X'. A variable substitution operation is performed to represent this set in terms of X.

RSchart Symbolic Representation

In this section a method for building a symbolic, OBDD representation of our Statechart variant, RScharts, is presented. Using this representation of an RSchart-modeled system it is possible to exhaustively verify system properties hold for all system behaviors. Non-determinism and all RSchart features are supported. A hierarchical encoding algorithm is presented and symbolic representation scaling issues are addressed.

Symbolic Representation of "State Configurations"

In order to represent our finite-state transition RSchart model with a Boolean representation, we must assign an encoding to all states and events present in a model. Recall from Chapter III, for an RSchart model, a system behavior step progresses the system from one global configuration to the next. In the end, we must have a unique encoding for every possible global configuration of the RSchart. For an RSchart model, the number of possible state configurations is bounded by the product of its concurrent sub-RScharts. The number of global configurations is bounded by |state configurations|×|2^{Events}|, making explicitly assigning encodings to the global configurations not feasible for large models.

A hierarchical state encoding scheme was chosen based on work presented in [47] to assign an encoding to all possible state configurations. The main requirement of this encoding scheme to be able to select the encodings for concurrent elements such that they may be composed to yield a encoding for an entire state configuration, avoiding an explicit assignment of encodings to state configurations. In the end, each valid state configuration (and global configuration) can be uniquely identified.

Properties of encoding:

- All children (sub-states) inherit their parent's encoding
- Each AND state has a number of sub-hierarchies, each with an OR state as its root. The set of encoding variables are partitioned such that each of these sub-

hierarchies receives a separate set of encoding variables, i.e. encoding variables are not shared among these sub-hierarchies.

- Children of an OR state receive unique encodings appended to the encoding inherited from their parents

The encodings are assigned in a top-down fashion with respect to the state hierarchy. Total number of bits needed to encode the states is calculated using a function NB(S) that specifies the number of variables needed to encode the state hierarchy of S.

$$\mathrm{NB(S)} = \begin{cases} \lceil \log 2(|H(\mathrm{s})|) \rceil + \max\limits_{\lambda \in H(s)} (\mathrm{NB}(\lambda)) & \text{if S is an OR state} \\ \sum\limits_{\lambda \in H(s)} (\mathrm{NB}(\lambda)) & \text{if S is an AND state} \\ 0 & \text{if S is a BASIC state} \end{cases} \tag{5}$$

For example, consider the statechart in Figure 11 and its corresponding state encodings in Figure 12. State X1 has a state encoding of 000xx, Y1 has a state encoding of 00x00. The 3rd bit position represents the unique encoding of the state of A1, the 4th and 5th bit positions represent the unique encoding of state of A2. A legal state configuration is {X1, Y1}. This state configuration can be represented by the conjunction of the two Boolean functions representing these two elements, also corresponding to a single state encoding of 00000.

A global configuration encoding includes the state configuration encoding and the event set encoding. Since set of event elements we need to represent is 2^E, where E is the set of all events, events are simply assigned a single variable, or bit position, in the overall global configuration encoding.

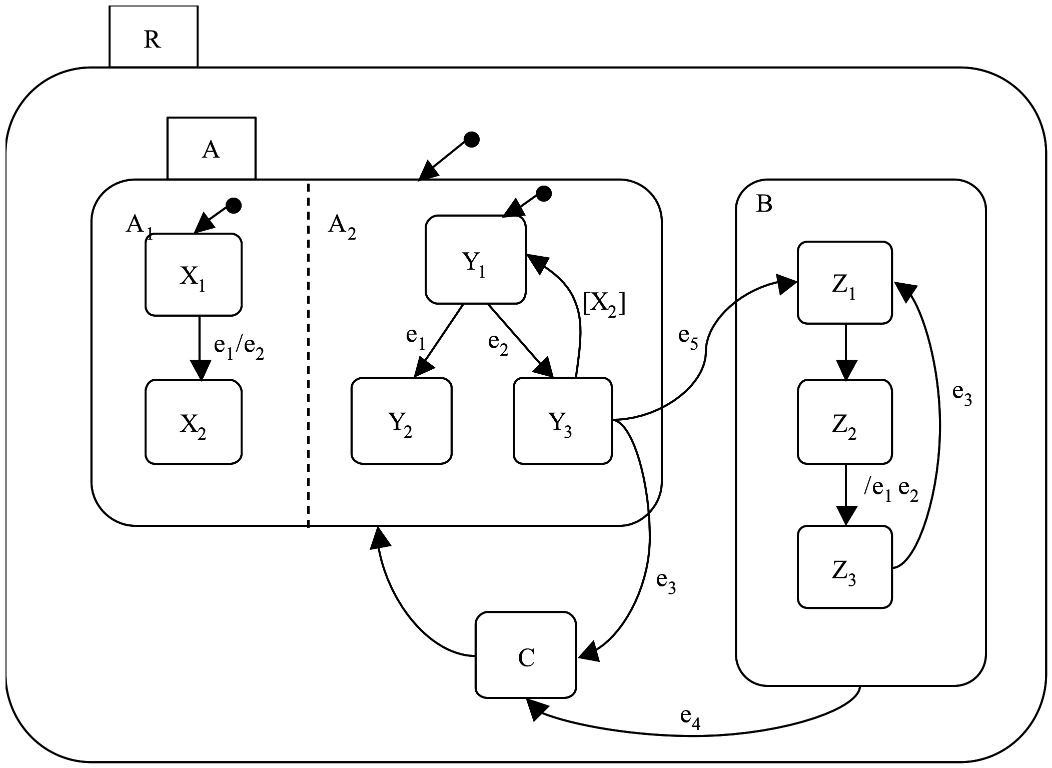


Figure 11: Example RSchart

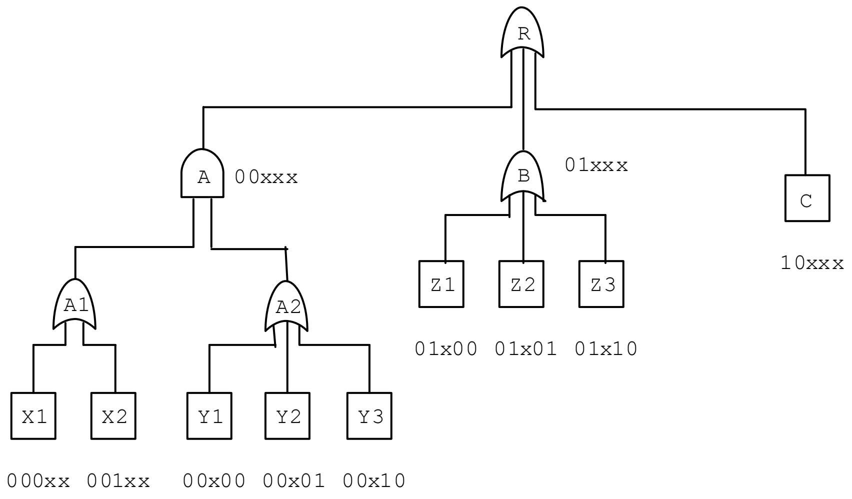


Figure 12: State encodings for the RSchart model in Figure 11

Creation of Transition Relation

The entire RSchart model can be represented by a single transition relation function that represents an equivalent "flattened" version of the model. RSchart features such as triggering conditions, guarding conditions, and implied transition priorities due to hierarchy must be supported. Non-deterministic behavior, if modeled, must be represented. The following procedure is used to construct the next state relation function.

A symbolic expression is created for each transition modeled:

For each $t \in T$, create

$$\text{trans}(t) = \text{source}(t) \cup \text{guard}(t) \cup \text{trig}(t) \rightarrow \text{dest}(t) \cup \text{action}(t) \qquad (6)$$

where, T is the set of all transitions in an RSchart model.

The functions source(t), guard(t), trig(t), dest(t), and action(t) return the RSchart components (states and events) associated with a modeled transition. These are defined in Chapter III.

For each $s \in S$, create

$$T_s(s) = \bigcup_{t \in T} trans(t), \text{ where } source(t) = s \qquad (7)$$

Ultimately, we will use $T_S(s)$ to compose a global transition relation for an entire RSchart model. The following issues must be addressed in the creation of our transition relation, TR, for it to accurately represent an RSchart model:

- Implicit transitions,
- Resolution of transition destinations, and
- Transition priorities.

Implicit Transitions

In the case that the system is in a BASIC state *s* and no transitions from that state are enabled to fire on the system step, then the next state of the system should again be that state, as implied by RSchart model semantics. This "null reaction" produces no events. In order for our transition relation to represent the correct next-state behavior of our system, this behavior must be captured as another transition in the executable model. For each BASIC state of the system an *implicit* transition with the proper firing conditions is created. Without this transition, if the system is in state *s*, but no transitions are enabled, the Image Computation will not return a corresponding next state for state *s*. We have just stated that if the system is in a BASIC state, then, regardless of the state of other system components or events, at least one transition must be enabled on the next system step. The same must be true as consider the RSchart model as a whole. As we compose a global transition relation for the entire RSchart, every valid global configuration, *c*, must have a corresponding transition relation in TR where *c* is the source configuration in one or more next-state relations in TR (i.e. the next-state behavior of every valid global configuration must be defined).

The computation of an implicit transition for a BASIC state is shown in Algorithm 1. We create an implicit transition for each BASIC state *s*. For each transition $t \in T_s(s)$ (all transitions originating from s) we compose $f_{leaving\_cond}$, representing the union of all *conditions* that can cause any $t \in T_s(s)$ to be fired. The firing condition for a transition t is $\{guard(t) \cup trig(t)\}$. We can now compute $f_{staying\_cond} = \neg f_{leaving\_cond}$ as set of all conditions that do not cause any outgoing transition to be fired. A new transition, $t_{implicit}$, is created and added to $T_S(s)$.

```
For all s $\in$ S with Type(s) = BASIC,
{
    $T_S(s)$ = {set of all transitions leaving state s}      // (defined in Equation 7)
    $f_{leaving\_cond} = \varnothing$
    For all t, where source(t) = s,
    {
        $f_{leaving\_cond} = f_{leaving\_cond} \cup \{ guard(t) \wedge trig(t) \}$
    }
    $f_{staying\_cond} = \neg f_{leaving\_cond}$
    $t_{implicit} = source(t) \cup f_{staying\_cond} \rightarrow source(t)$
    $T_S(s) = t_{implicit} \cup T_S(s)$
}
```

Algorithm 1: Implicit transition computation

Resolution of Transition Destinations

The RSchart semantic model supports the destination of a transition being a *superstate* (both AND and OR states are referred as superstates). A superstate alone, however, is not valid state configuration. Initial transitions are used to determine the actual BASIC states that are entered when a superstate is entered. For example, the statechart in Figure 13 has a transition from C to A. A1 has an initial transition to X1 and A2 has an initial transition to Y1. Therefore, the transition from C to A should transition from C to the legal state configuration (X1, Y1). Transitions to superstates must be resolved to the correct BASIC state(s).

Transition priorities

Transitions leaving superstates must be given priority over transitions that may occur within the superstate. If a transition leaving a superstate is enabled to fire, then the firing of transitions with that superstate must be suppressed. To accomplish this suppression, all transitions in the sub-hierarchy of this superstate must be amended with the condition that they will not fire if the transition leaving the superstate is enabled to fire. This behavior could be done by modifying the transition trigger expression, trig(t), and guarding condition, guard(t), before the symbolic representation of each transition is constructed in Equation (6). Instead, the choice was made to modify the OBDD representation of the appropriate transitions, the $T_S(s)$ of the states in the entire sub-hierarchy of the superstate. Neither solution has any significant advantage over the other.

This procedure is shown in Algorithm 2. For each transition that has a superstate, *s*, as a source state, we compute the firing condition of this transition f_{FC} = guard(t) $\wedge$trig(t). We compute the suppression condition $f_{SC} = \neg f_{FC}$, to be applied to each set of $T_S(s)$ transitions for all states in the subhierarchy of the superstate *s*.

```
For all s $\in$ S with Type(s)={AND, OR}
{
    For all t, where source(t)=s,
    {
        $f_{FC}$= guard(t)$\wedge$trig(t)
        $f_{SC} = \neg f_{FC}$
        For all $\lambda \in H_{Sub}(s)$, where $H_{Sub}(s)$ contains all states in the sub-hierarchy of s
        {
            $T_S(\lambda)=T_S(\lambda)\wedge f_{SC}$
        }
    }
}
```

Algorithm 2: Prioritizing transitions

Creation of RSchart Transition Relation

Finally, the translation relation, TR(S), is created by hierarchically composing the set of transitions for each state, $T_S(s)$, as shown below.

$$TR(S) = \begin{cases} \bigcup_{\lambda \in H(\mathrm{S})} TR(\lambda) \cup T_S(S) & \text{if S is OR} \\ \bigcap_{\lambda \in H(\mathrm{S})} TR(\lambda) \cup T_S(S) & \text{if S is AND} \\ T_S(S) & \text{if S is BASIC} \end{cases} \tag{8}$$

The same Image function introduced in (4) is used to traverse the state space defined by TR(s) just as for the flat FSM case.

Distributed Statechart

The size of the OBDD created from the operation in Equation (10) can, for models with a high degree of concurrency, exhibit memory problems. This is due to the composition of concurrent elements via a cross product of their spaces (transitions are implicitly created for all possible global configuration to global configuration transitions). If we could avoid the composition of these product spaces then the representation size of the system transition relation could be greatly reduced.

Burch, in [48], presented a method for using partitioned OBDD transition relations to represent synchronous and asynchronous state transition graphs. This method significantly lowered verification times and overall system OBDD representation sizes. In a similar manner, the idea of partitioned transition relations has been adapted for use with the hierarchical structure of RScharts.

Let's take a look at the Image Computation function introduced in Equation (4). It is clear from the construction of TR(s) in Equation (10), the transition relation can be replaced by its distributed equivalent shown below.

$$\mathrm{TR}(S_{\mathrm{ROOT}}) = \mathrm{TR}(S_{\mathrm{ROOT}}) \vee (\mathrm{TR}(S_1) \wedge \mathrm{TR}(S_2)) \ldots \qquad (9)$$

The distributed Image Computation may be performed without ever computing the monolithic TR(X, X').

$$\mathrm{Image}\big(\mathrm{S}(X)\big) = \Big(\exists_X \big(\mathrm{S}(X) \wedge \big(TR_{Root}(X,X') \vee \big(TR_{S1}(X,X') \wedge TR_{S2}(X,X')\big)\ldots\big)\big)\Big)\Big|_{X' \to X} \qquad (10)$$

The distributed representation must be such that the variables representing the next state relations for these orthogonal areas are partitioned. Each AND state present in the RSchart hierarchy will also be present in the distributed transition relation. A minimized statechart hierarchy, $H_{MIN}(s)$, is created to capture the structure of the distributed transition relation. The structure, $H_{MIN}(s)$, is derived from the statechart hierarchy according to the following rules:

- All orthogonal areas are represented with a separate variable partition as to avoid the computation of the product spaces of the orthogonal elements of the statechart. AND states must retain the same number of direct-descendant sub-states.
- If an OR node has no AND descendants then its entire hierarchy is represented by a single transition relation.

A sample RSchart and its state hierarchy are shown in Figure 13 and Figure 14, respectively. Graphically, Figure 15 shows the distributed transition relation, $H_{MIN}(s)$, derived from the state hierarchy. Notice the concurrent spaces of the RSchart always have separate transition relations for both the sub-hierarchies of state A and state J. Functions have been created to perform standard OBDD operations on the distributed representation as though it were a single OBDD.

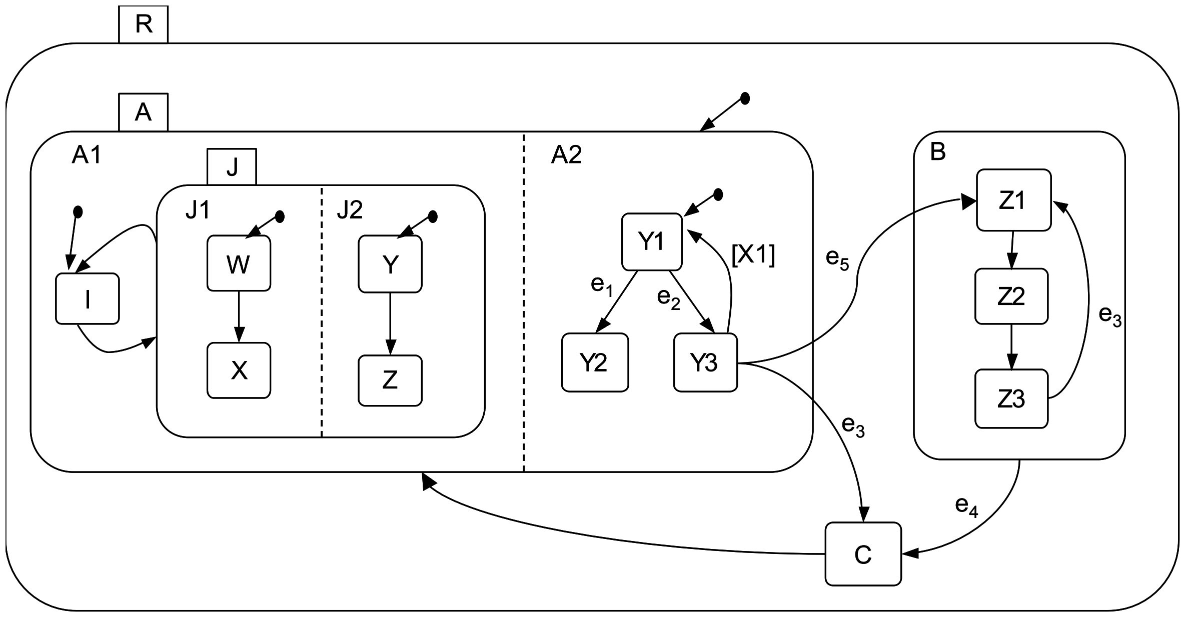


Figure 13: Sample RSchart

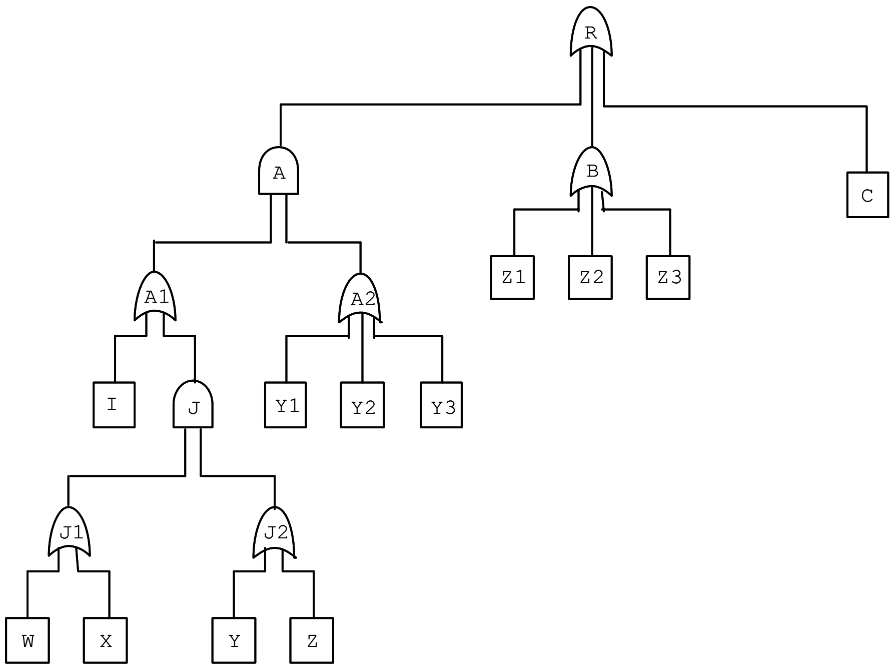


Figure 14: Hierarchy of RSchart in Figure 13

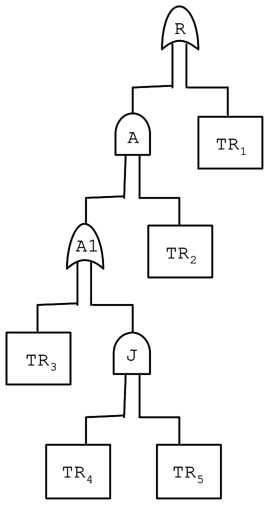


Figure 15: $H_{MIN}(s)$ Minimized RSchart hierarchy

For models with a monolithic transition relation that could grow past the bounds of physical memory, this approach results in several orders of magnitude decrease in computation time due to the transition relation being kept in physical memory. For systems with some fairly small component transition relations, it may be more efficient to perform the appropriate conjunctions/disjunctions to compose the smaller set of relations into a single transition relation.

Currently, this method is only used for representation of the transition relation of an RSchart although a partitioning scheme such as this could also be used to reduce the overall OBDD representation size for state sets as well. In a later paper by Chan, et al. [49] work is discussed that shows how to modify SMV to use a similar conjunctive/disjunctive partitioning method to reduce the size of the OBDD representing state sets and also the transition relation.

Timed Statechart Symbolic Representation

A timed extension of RScharts has been developed, Timed Restricted Statecharts (TRScharts). TRScharts are a synchronous finite state model similar to statecharts with the addition of a discrete time semantics. In TRScharts, each state must be active for a time duration *t* before the system may progress to another state. This time duration *t* may be specified directly or specified non-deterministically from an interval $[t_{lower}, t_{upper}]$. Specifying an interval time defines a non-deterministic behavior where the actual time duration of the state is chosen non-deterministically from this interval, where $t_{lower} \leq t_{duration} \leq t_{upper}$.

Since TRScharts implements a synchronous semantics that can be described by a finite-state transition system, TRScharts can be represented by the same OBDD symbolic representation methods described for RScharts. In order to do this, however, a state-expansion technique must be used to represent the time progression of a state as its duration elapses. Any TRSchart model may be mapped to an equivalent untimed model where each step represents one unit of time. Figure 16 is a TRSchart with its corresponding behavior trajectories. Figure 17 is a finite-state model equivalent to the TRSchart model in Figure 16. The untimed states are labeled with a subscript indicating the number of steps the system has been present in a particular state, where the state corresponds to the TRSchart state from which it was derived.

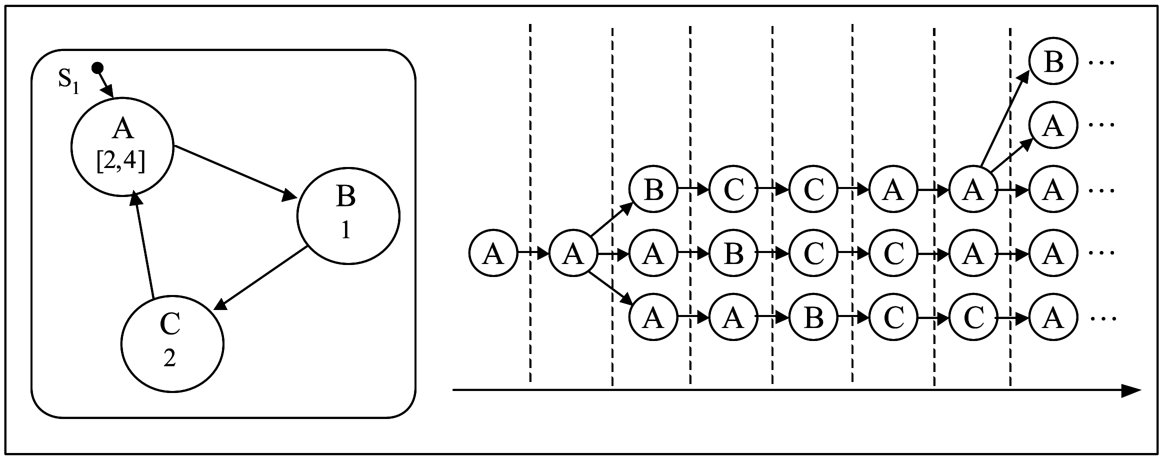


Figure 16: TRSchart and its corresponding behavior trajectories.

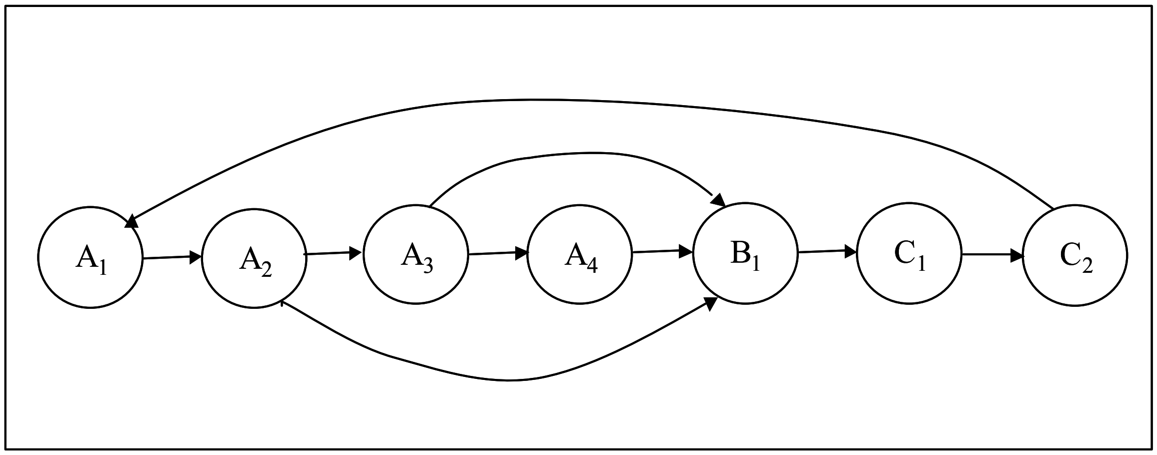


Figure 17: Untimed model equivalent to the TRSchart in Figure 16

Using MTBDDs to Represent Timed Transition Systems

We would like to establish a more efficient symbolic representation for the TRSchart timed finite-state model than simply performing a conversion to an untimed domain. To accomplish this, extending the original set membership function to directly associate an integer with each element of the set was investigated. A state's time progression (ultimately a global configuration's time progression) will be denoted by this integer. The new set membership function is represented by $f_T(s)$. MTBDDs will be used to associate an attribute with each state to indicate its *time progression*, similar to a method presented by Ruf and Kropf in [60]. Just as with the standard set membership function, if the function returns a null value for a set then that element is not present in the set.

$$f(s) = \begin{cases} 1 \text{ if } & s \in S \\ 0 \text{ if } & s \notin S \end{cases} \tag{11}$$

$$f_T(s) = \begin{cases} N \text{ if } & s \in S \\ 0 \text{ if } & s \notin S \end{cases} \text{, where N is a positive integer} \tag{12}$$

The set member function $f_T(s)$ can be represented with a MTBDD. An MTBDD is a generalization of an OBDD in that now the value of a terminal node is not limited to {true, false} but may be any attribute, restricted here to a positive integer value. An MTBDD can represent a multi-valued Boolean function (sometimes called pseudo-Boolean function) that associates some value with each evaluation. Where a standard Boolean function can be thought of as a function that associates a true or false result with every possible evaluation, a multi-valued Boolean function associates some value with each outcome [17].

$$f\text{: } \{1, 0\}^n \rightarrow N \text{, where N is a positive integer} \tag{13}$$

From Chapter III, recall a TRSchart "state" is represented by the pair $\langle s, t \rangle$, where s is a BASIC state and t is that state's current time progression value, where $0 \leq t \leq t_{duration}$, and $t_{duration}$ is non-deterministically chosen from the state's time interval range ($t_{lower} \leq t_{duration} \leq t_{upper}$). This pair is referred to as a *timed state*. The set membership function, $f_T(s)$ in Equation (14) must be capable of representing *all* possible *sets* of timed states. This includes representing multiple timed states that have the same BASIC state s, but a different time progression value. The set membership function, $f_T(s)$, must be able to associate multiple time progression values (via its attribute) with a given state *s*.

The attribute that, $f_T(s)$, associates with a state a positive integer value N. This value represents a time progression value encoded using a one-hot encoding scheme, where each bit position represents the presence of a state in a stage of its time progression. The integer attribute of each state is defined as a bit vector B: $b_n \ldots b_1$, where $n = t_{upper}$, the maximum time progression value we need to represent for this state. A "one" value for a bit position b_x represents the presence of state S in a time progression of $\langle S, x \rangle$. This time progression value is a local procession value for each state.

Consider the TRSchart in Figure 16. A multi-valued function is created to represent the set $S_T = \{\langle B, 1\rangle, \langle C, 1\rangle, \langle C, 2\rangle\}$. Consider a membership function $f_T(s)$ representing a timed state set S_T. We would like an attribute to be associated with the satisfying evaluation of $f_T(s)$ representing state B and its time progression $\langle B, 1\rangle$ and state C at its two states of time progression: $\langle C, 1\rangle$ and $\langle C, 2\rangle$. The membership function $f_T(s)$ is shown below:

$$f_T(s) = \begin{cases} \text{"000001"} & \text{if } s = B \\ \text{"000011"} & \text{if } s = C \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

The set union and intersection operations are defined for the MTBDD set membership functions. Recall for the OBDD set membership functions, set union and intersection operations were implemented via Boolean OR and AND operations, respectively. In a similar manner, analogous MTBDD operations are defined. The operations are implemented using the basic *apply* function for MTBDDs (shown in Algorithm 3). The MTBDD *apply* function is shown below for the set UNION operation.

It is virtually identical to the OBDD *apply* OR function except that in the MTBDD case the terminal nodes of F and G are constant values other than {true, false}. A terminal operation is used, just as in the OBDD case, to define the operation. The *apply* function recursively computes a new graph representing $F \cup G$. Before recursively calling the computation of two sub-trees of F and G, the algorithm checks to see if this computation has been performed before and if its result is stored in the OperationCache. If so, the cached result is returned and that branch of the recursion ends. When terminal nodes of both F and G are reached, the terminal operation *UnionTerminalOp* is performed computing the OR of the value of the terminal nodes (in this application the terminal nodes are bit vectors, represented by B_F and B_G). Set Intersection is computed in a similar way using a terminal operation that is the AND of the bit vectors.

```
Apply(F, G, UnionTerminalOp)
{
    if ( F is constant && G is constant )
    {
        val = UnionTerminalOp(value(F), value(G));
        construct a terminal node result with value(result) = val
        return result;
    }
    else if (F, G, UNION) $\in$ OperationCache
    {
        return OperationCache(F, G, UNION);
    }
    else
    {
        $x_i$ = the next variable both F and G depend on (i is the index of this variable)
        T = Apply($F|_{xi=1}$, $G|_{xi=1}$, UNION);
        E = Apply($F|_{xi=0}$, $G|_{xi=0}$, UNION);
        construct a new node result at index i with high(result) = T and low(result) = E
        CacheInsert(F, G, UNION, result);
        return result;
    }
}

UnionTerminalOp ($B_F$, $B_G$)
{
    construct a terminal node result with value = $B_F \vee B_G$;
    return result;
}
```
Algorithm 3: MTBDD Apply function [16]

Creation of Transition Relation

The transition relation for TRScharts is constructed in a similar manner as the untimed case with the addition of an attribute assigned to each next-state transition mapping. The attribute in this case specifies the initial time progression value that is the duration of the timed state(s) into which we will transition. The time duration of the next-state may be a single value or a range of time values. The transition attribute values are encoded in a one-hot fashion similar to that of the augmented set membership described previously.

R: $(S \times S') \rightarrow B$, where B is a bit vector

$$b_0 = \begin{cases} 1 & S \rightarrow S \\ 0 & S \rightarrow S' \end{cases}$$

$b_1 = 0$

$b_{n+1} .. b_2$ = bit vector identifying in initial clock value

where n = maximum t_{upper} value of all states

To simplify the Image Computation presented in the next section, the time progression value in the next-state relation is "pre-shifted". The time value vector is left-shifted one position, always leaving a zero in the b_1 position. This will be discussed further in the next section.

For the modeling of timed systems, there is a need to have a relation that represents the case that a state's time duration has not yet elapsed and the system is not allowed to transition from that state to a new state. That is, the system must stay in its current state. To differentiate this new type of relation, an identification bit is added to the next-state transition relation attribute. The b_0 position of the attribute bit vector B was chosen to identify a relation as either: 1) transition identifying a next-state mapping (*next-state transition*) or 2) a transition from state *S* to the same state *S*, the case where a state's time duration has not elapsed and the system is not yet allowed to leave that state (*non-elapsed transition*). This is analogous to the transition inserted in the RSchart case to create the implicit behavior: "if no modeled transitions are enabled from the current state, then the system should remain in that state". The relation enforces that for every possible global configuration the transition relation will always produce a complete next global configuration set of the system.

Both the *next-state transitions* and the *non-elapsed transitions* relations must co-exist in a single global transition relation. These two types of transitions may overlap (one relation may be the same as/a subset of the other) causing the attributes of the relations to become a single attribute. The two types of transitions must be differentiated by the introduction of a new variable, x_0. All *non-elapsed transitions* will have this variable, x_0, AND'ed to the function representing its relation; $\neg x_0$ will be AND'ed to each *next-state transitions* function.

The method of timed-finite state representation has been extended to represent a possible range of times a state must be occupied before the system may transition to another state. This time interval associated with a state represents the state's time duration will be a value within this range, $t_{lower} \leq t_{duration} \leq t_{upper}$. These time ranges ($t_{lower,}$ t_{upper}) define non-deterministic behavior (the degree of non-determinism increases with the size of t_{upper} - t_{lower}). In the case of a time interval the system may non-deterministically transition into a set of timed states; each the same state with a different initial time progression value.

An example demonstrating the behavior of a TRSchart model with interval times associated with a state is shown in Figure 18. In the function $t_{AB}(s, s')$ bits b_5, b_4, b_3 are set to indicate the initial time progression value of the destination states. In this case, the destination is a set of states: $\{\langle B, 4\rangle, \langle B, 3\rangle, \langle B, 2\rangle\}$. In the untimed model, the equivalent set of states *State A* will transition into is indicated by the states enclosed in the dashed area. An equivalent finite-state representation is shown to demonstrate the behavior of the TRSchart S_1.

$t_{AB}(s, s') = \neg s_1 \neg s_0 \wedge \neg s_1' s_0' \rightarrow$ "111000"

$t_{BC}(s, s') = \neg s_1 s_0 \wedge s_1' \neg s_0' \rightarrow$ "001000"

$TR_T(s, s') = t_{AB} \vee t_{BC}$

State	s_1 s_0
A	0 0
B	0 1
C	1 0

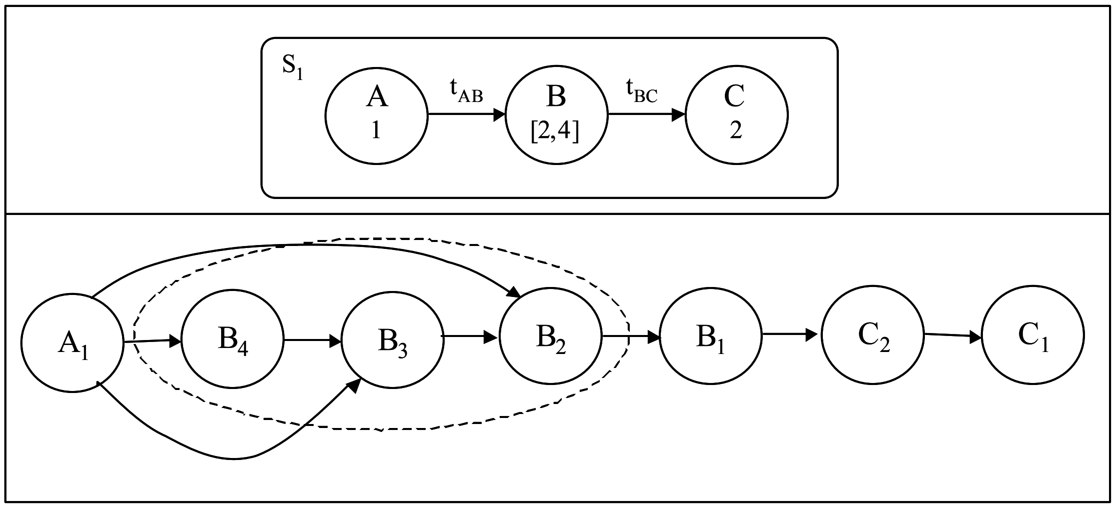


Figure 18: TRSchart with time intervals and associated (finite-state) behavior

MTBDD Image Computation

An image computation procedure has been developed to find the next state of a TRSchart- modeled system, given the system transition relation, $TR_T(s, s')$. The Image Computation function should implement the following behavior: If a state's duration has elapsed, the system is allowed to exit that state. The system's next-state will be determined by its corresponding next-state relation in the transition relation $TR_T(s, s')$. If a state's duration has not elapsed, then the system will remain in that state with the state's clock timer being updated to reflect the passage of one unit of time.

This method of computing the image computation is implemented as an MTBDD algorithm. The implementation of this behavior first computes the corresponding next-state relations from our current set of states. This function is implemented recursively via the general MTBDD apply function with *ForwardStepTerminalOp* as the terminal operation applied.

$$S_{NEXTREL}(s, s') = \text{Apply}(S_{CUR}(s), TR_T(s, s'), \text{ForwardStepTerminalOp})$$

When a terminal node of both F and G is reached, the terminal function *ForwardStepTerminalOp* is applied. A non-zero terminal node of the transition relation MTBDD (TR_term, also labeled as input B_F) represents two classifications of relations. The relation this terminal is associated with is a *next-state transition* or the relation is a *non-elapsed transition*. A non-zero terminal node of the current state set MTBDD (CS_term, also labeled as input B_G) represents the state of the time progression of the current state set. The terminal operation logic is as follows: If a CS_term has a bit set in bit position b_0 indicating that there are timed states in this timed state set whose duration has elapsed and must transition, and the corresponding terminal node value from the transition relation (TR_term) is associated with a *next-state transition*, then return TR_term. If CS_term has bits set in the $b_{n+1}..b_2$ positions then the intermediate states represented by these time progression bits will stay in the same main state on the next cycle. The system must stay in this state until its duration elapses and the corresponding terminal value from the TR function is a “non-elapsed” transition, and then return the CS_term value.

```
ForwardStepTerminalOp($B_F$, $B_G$)
{
        TR_term = $B_F$;
        CS_term = $B_G$;
        trans_id = TR_item [0];
        elapsed_bit = CS_term[0];
        non-elapsed_bits_present = non_elapsed $\neg$(0x1) $\wedge$ CS_term;
        // if real transition and an elapsed state is present
        if(trans_id = 0 and elapsed_bit = 1)
                new_value = TR_term;
        // if "non-elapsed" transition and non-elapsed states present
        else if (trans_id = 1 and non-elapsed_bits_present)
                new_value = CS_Term;
        else
                new_value = $\emptyset$;
        return new_value;
}

// NOTE: lsb bit in state set representation is not valid and is not used. (this is the trans_id bit)
```

Algorithm 4: Forward Step Terminal Operation

This function replaces the Boolean AND operation used in the untimed OBDD case. This operation could be though of as a "selective" AND operation, where the traditional AND result of two terms occurs selectively based on the attributes of the two operands.

Now, we have the selected relations, $S_{NEXTREL}(s, s')$, containing the next-state of the system in the *s*' set of variables. The following computations are performed just as in the untimed OBDD case:

$S_{NEXT}(s') = \exists_{s,\, x0}(S_{NEXTREL}(s, s'))$

$S_{NEXT}(s) = S_{NEXT}(s')|_{s' \rightarrow s}$

$S_{NEXT}(s) = \text{ZeroEventsNotPresent}(S_{NEXT}(s))$

The last step in the Image Computation is the *TimeTick* operation. *TimeTick* serves to decrement the "time progression clock" attributes of all timed states in a set by one unit of time. To perform the decrement, only a left shift operation of the Boolean attribute vector is needed. Note the reason all the initial time values in the transition relation attributes were pre-shifted is so that the TimeTick function can uniformly decrement the time progression clocks of both the states that remain unchanged from the last cycle and the new state we have just transitioned into. The implementation of TimeTick is performed via the MTBDD apply1 function. The apply1 function operates on a single MTBDD. Similar to the two-operand general *apply* function, *apply1* recursively traverses an MTBDD until a terminal node (attribute) is reached. In this case a single argument terminal function (*TimeTickTerminalOp*) is used to perform the decrement operation.

$S_{NEXT}(s) = \text{apply1}(S_{NEXT}(s), \textit{TimeTickTerminalOp})$

The implementation of *TimeTickTerminalOp* is shown in **Error! Reference source not found.** below.

```
TimeTickTerminalOp(B_F)
{
        return (B_F >> 1)   // The bit vector B_F is right-shifted by 1 bit
}
```

Algorithm 5: Time Tick Terminal Operation

Concurrency in Timed Finite State Representation

The timed finite state representation method presented thus far is not sufficient to represent a state configuration composed of the states of multiple concurrently operating timed finite state systems in the same manner as for the untimed case. Presented below are extensions necessary to represent concurrent system model TRScharts. As with RScharts, a global configuration (global "state" of the system) will be represented with a single function (in this case, a multi-valued Boolean function). The current time progression value must be captured for each basic state within a global configuration.

Just as the state encodings for RScharts are chosen such that a set of variables is assigned to represent each concurrent state space of the model, an analogous assignment is performed for the partitioning of the "time progression" bits. The attribute vector, B, is partitioned into sub-vectors, one sub-vector per concurrent state space. The total number of bits needed is computed using the state hierarchy function shown below.

$$NTB(S) = \begin{cases} \max(NTB(\lambda)) + NTB(\chi) \text{ for all } \lambda \in H(\mathrm{S}), \text{ where } \mathrm{Type}(\lambda) \neq AND, & \\ \qquad \text{for all } \chi \in H(\mathrm{S}), \text{ where } \mathrm{Type}(\chi) = AND & \text{if S is OR} \\ \sum (NTB(\lambda) + 2) \text{ for all } \lambda \in H(\mathrm{S}) & \text{if S is AND} \\ \text{time\_upper(S)} & \text{if S is BASIC} \end{cases}$$

where time_upper(s) is a function that returns t_{upper} of state s.

With the representation scheme we have outlined thus far, a single multi-valued Boolean set membership function is unable to represent all valid global configurations (that is, all state configurations at all possible clock progression values of each state in the state configuration). For example, we may want to represent a set of global configurations $\{S_{GC1}, S_{GC2}\}$ with $S_{GC1} = \{\langle A, 1\rangle, \langle B, 2\rangle\}$ and $S_{GC2} = \{\langle A, 2\rangle, \langle B, 1\rangle\}$ via a set membership function where state A is encoded with the term "s_3s_2" and state B is encoded with the term "s_1s_0".

$\{\langle A, 1\rangle, \langle B, 2\rangle\}$: $f_{GC1}(s) = s_3s_2\ s_1s_0 \rightarrow$ "00100 01000"

$\{\langle A, 2\rangle, \langle B, 1\rangle\}$: $f_{GC2}(s) = s_3s_2\ s_1s_0 \rightarrow$ "01000 00100"

Taking the union of these two state configurations using the *UnionTerminalOp* operation outlined previously, the result would be:

$f_{GC}(s) = \mathrm{Apply}(\ f_{GC1}(s), f_{GC2}(s), \mathit{UnionTerminalOp}\ )$

$f_{GC}(s) = (s_3s_2\ s_1s_0 \rightarrow$ "00100 01000"$) \vee (s_3s_2\ s_1s_0 \rightarrow$ "01000 00100"$)$

$f_{GC}(s) \neq s_3s_2\ s_1s_0 \rightarrow$ "01100 01100"

This result is incorrect. According to the encoding scheme, the function $f(s) = s_3s_2\ s_1s_0 \rightarrow$ "01100 01100" represents $S_1 \times S_2$, where S_1 and S_2 represent the concurrent spaces of our model.

$S_1 \times S_2 = \{\langle A, 1\rangle, \langle B, 2\rangle\} \times \{\langle A, 2\rangle, \langle B, 1\rangle\}$

$= \{\langle A, 1\rangle, \langle B, 1\rangle\}, \{\langle A, 1\rangle, \langle B, 2\rangle\}, \{\langle A, 2\rangle, \langle B, 1\rangle\}, \{\langle A, 2\rangle, \langle B, 2\rangle\}$.

It is now clear that the MTBDD representation as defined so far is not able to capture all *sets* of all possible global configurations for the timed model in a single multi-valued function. The representation is augmented to support multiple attributes with which an evaluation of $f_{GC}(s)$ may be associated.

The following could be interpreted as a correct representation of $S_{GC1} \cup S_{GC2}$:

$f_{GC}(s) = s_3s_2\ s_1s_0 \rightarrow$ {"00100 01000", "01000 00100"}

A modification had been made to the MTBDD representation itself to support the attributes defining a set of Boolean vectors. Our multi-valued Boolean function may map a Boolean evaluation onto a set of *m* Boolean vectors. The set membership function becomes $f_{TA}(s) \rightarrow \{B^n\}^m$. The membership function now defines a "set of sets" where satisfying values of $f_{TA}(s)$ are mapped to a set of elements.

The *UnionTerminalOp*, *ForwardStepTerminalOp*, *TimeTickTerminalOp*, *IntersectionTerminalOp*, and other relevant functions have been modified to support operations on MTBDDs with set attributes. For example, the *UnionTerminalOp* terminal function now, when terminal nodes of both operands F and G are reached, takes the union of the set of attributes of each terminal node reached and returns that result.

```
UnionTerminalOp($SB_F$, $SB_G$)  // $SB_F$ and $SB_G$ are sets of attributes
{
        return $SB_F \cup SB_G$;
}
```

Algorithm 6: Set Union Terminal Operation

Although a single function, $TR_T(s, s')$ may represent the entire system behavior, to avoid the full product-space construction the transition relation is constructed using the distributed transition relation method as described for the untimed RSchart case. The global transition relation should be constructed to guarantee to have a corresponding next-state for *every* valid global configuration - just as the RSchart next-state relation was shown to have earlier.

The key operation in performing an image computation, *ForwardStepTerminalOp()*, performs the same operations as in the non-concurrent case, but now the terminal operation is applied to each sub-vector of each member of a terminal set attribute. When the image computation is performed on a TRSchart model with concurrency present, time is progressed simultaneously in all concurrent spaces. The *ForwardStepTerminalOp()* uniformly performs a time progression for all time progression sub-vectors of all concurrent spaces present in the model.

Testing for Set Membership

Symbolic model checking algorithms exploit the fact that OBDDs work well for representing large sets of elements. As such, all operations generally operate in terms of sets. Occasionally though, there are situations where it may be necessary to find out what elements are contained in a set. Testing for set membership may not be possible for a complex RSchart that has an exponential number of state configurations with respect to the number of labeled states. In general, it is not feasible to enumerate through all state configurations to check for inclusion in the OBDD state set.

A method for identifying all configurations in an OBDD-represented set has been developed that avoids this enumeration. First, the OBDD that represents the system state set is decomposed into a sum-of-products representation derived directly from the OBDD graph itself by recording satisfying paths traversed from the root to the *true* terminal. Due to possible “don’t care conditions” within the SOP term, each SOP may represent a set of global configurations of its own. The encoding of each SOP term is used to make decisions of how to traverse the state hierarchy tree until the leaf nodes (BASIC states) of the tree are reached identifying the global configuration(s) represented. The global configurations represented by each SOP term are collected and returned as the result. This method for testing for set membership works equally well for MTBDD-based representation.

Check for Non-determinism

For a complex model, it may be desired to have a function to check that the modeled system is deterministic. If every valid global configuration has, at most, one transition that may be enabled, then the system is deterministic. A method was developed to check for a deterministic model. Although this could be done on the model itself, the method below operates directly on the TR(s, s’) representation of the system. Remember that the transition relation, TR, contains not only all modeled transitions, but also implied transitions (“stay transitions”) and transition priorities have been enforced where appropriate. To check the TR itself has the advantage that the test not only checks that model does not contain non-deterministic behavior, but also checks that no non-deterministic behavior has been inadvertently created through the synthesis process of the TR.

The test for determinism is detailed in Algorithm 7. The algorithm tests for local deterministic operation from each BASIC state. For each BASIC state *s*, an *and* operation is performed with the transition relation, TR to find all transition mappings with state *s* as a source state. The set of all trigger conditions, *trigger_cond_set*, is found for this set of transitions. This set of triggers is decomposed into a sum-of-products form where all terms of this expression are compared with each other to check for intersection between the sets. If an overlap exists, then there are two or more transitions originating from this state, thus there is non-determinism.

Note is it possible to have a model that tests as non-deterministic, but is deterministic in operation. This would be the case if orthogonal component(s) are non-deterministic, but, by design, the resulting behavior of the composition of these components is deterministic.

```
Boolean Deterministic()
{
    For all states $s \in S$ where Type(s) = BASIC
    {
        // Let $x_s$ be present state variables, $x_e$ be present event variables
        // Let $x_s$' be next state variables, $x_e$' be next event variables
        bdd trigger_cond_set = $\exists$ $x_s$, $x_s$', $x_e$' ($s(x_s) \wedge TR(x_s, x_e, x_s', x_e')$);
        mat = return_SOP_matrix_form(trigger_cond_set);
        bdd tmp = $\emptyset$;
        for i = 1 to mat.num_rows()
        {
            // each row of matrix is a single term of the cond set from state s
            bdd SOP_term = mat_to_bdd(mat[i]);
            if((tmp $\wedge$ SOP_term) $\neq \emptyset$)
                return false;
            tmp = tmp $\wedge$ SOP_term;
        }
    }
    return true;
}
```

Algorithm 7: Test for Determinism

Summary

A method was presented for representation and traversal of RSchart models efficiently via a symbolic method, OBDD. Analogously, a symbolic MTBDD-based method was developed for the representation of TRScharts. The use of these methods will potentially allow model checking of very large state spaces. Reachability algorithms were implemented that operate on the symbolic OBDD/MTBDD representations for breadth-first exploration of system behavior. Finally, a method for testing for a deterministic model was presented.

CHAPTER V

SYMBOLIC ANALYSIS OF MULTIPROCESSOR REAL-TIME SYSTEMS

System verification is necessary for today's complex, embedded systems. Embedded real-time multi-processor systems exhibit complex interactions resulting in a very large behavior space. It is necessary to prove correctness of the *implementation* of these real-time systems for *all* possible system behaviors. A method for system verification is presented in this chapter that is targeted for embedded multi-processor systems whose processing is specified by a dataflow model. In this chapter we define our own dataflow model suited for this problem domain. It is called Real-Time Dataflow Model (RTDF).

Verification algorithms are presented in this chapter that fully explore the behavior space of the system models to verify that the system will always satisfy some property (such as no buffer overflow, or, at a higher level of specification, the system is "schedulable"). In order to accomplish this, a finite-state representation of the system is constructed. This FSM construction is shown in this chapter. From the system models we can derive a computation engine of the system that will allow us to explore system behavior. The behavior space of the system is generated and represented in the form of a TRSchart, a finite-state, discrete time model (presented in Chapter III).

With this representation, all possible behaviors that system may exhibit can be examined. Exhaustive verification is necessary to guarantee the model will meet its requirements. As a side benefit, these analysis techniques may also be used to find the overall extremes of system performance: maximum throughput, and the maximum occupied space of system buffers can be calculated.

The RTDF system models can have non-deterministic behavior. This is useful when modeling to test for worst-case behaviors. Where exact system behavior is not known, it is useful to model a range of behaviors. While critical for a pessimistic/conservative modeling and verification approach, the examination of a large range of behaviors does pose a problem for verification. Even systems with a seemingly small amount of variability can have an extremely large number of possible behaviors. This is especially true of concurrent systems. The key problem is: "How can we examine such a behavior space that even for reasonably small systems quickly grows to be unmanageable?". This is often referred to as the as the *state-explosion problem*. Symbolic model checking has shown promise in several application areas in tackling the types of problems where complete coverage of a state-space is required. Traditional simulation methods are simply not possible. Building on symbolic techniques presented in Chapter IV, this chapter presents a method for exhaustive examination of this potentially very large space providing a proof of correct system operation correct in all cases.

Described in this chapter are the models used to specify our system. Next the creation of the finite-state behavior model is described. Finally, verification algorithms are presented.

Embedded System Modeling

A well-defined model of the system is essential to provide the opportunity for automated analysis techniques. In this chapter the *model of computation* is defined for the specification of a dataflow program that will be implemented on a set of physical hardware processing resources. A model is also defined for the specification of the hardware resources of a multi-processor embedded system on which the dataflow computation is implemented.

The dataflow computation is well suited for the problem domain of embedded signal-processing systems. The dataflow representation itself is very intuitive; it closely resembles the type of sketch a DSP engineer would create to specify an algorithm implementation. This modeling approach also allows the algorithm designer to begin the process of design and analysis of the system functionality while remaining isolated from the hardware-specific implementation details (scheduling, inter-processor communication, etc.).

The dataflow model of computation is a functional description rather than a procedural one allowing concurrency to be explicit in the computational model. This is in contrast to a procedural algorithm description where the order of execution can be over-constrained and data dependency information can be difficult to retrieve after it has been lost in a procedural mapping.

The following sections describe the Real-Time Data Flow model and the Resource Model upon which the RTDF model of computation is implemented. These two models will be used to fully describe a system implementation.

Resource Model

The Resource Model defines a multi-processor network composed of the system processing elements and the communication topology. The communication structure is a set of point-to-point, uni-directional communication links (sometimes referred to in this thesis as "commlinks").

The Resource Model is defined by $R = \langle P, L \rangle$, where P is the set of processing elements and L is the set of communication links connecting processing elements where $L \subseteq (P \times P)$. Each communication link has a *fifo depth* attribute. The fifo depth defines the length of the queue contained in a communication link. Communication links, L, have a fifo depth mapping function $f_d : L \rightarrow N$, where N is a non-negative integer.

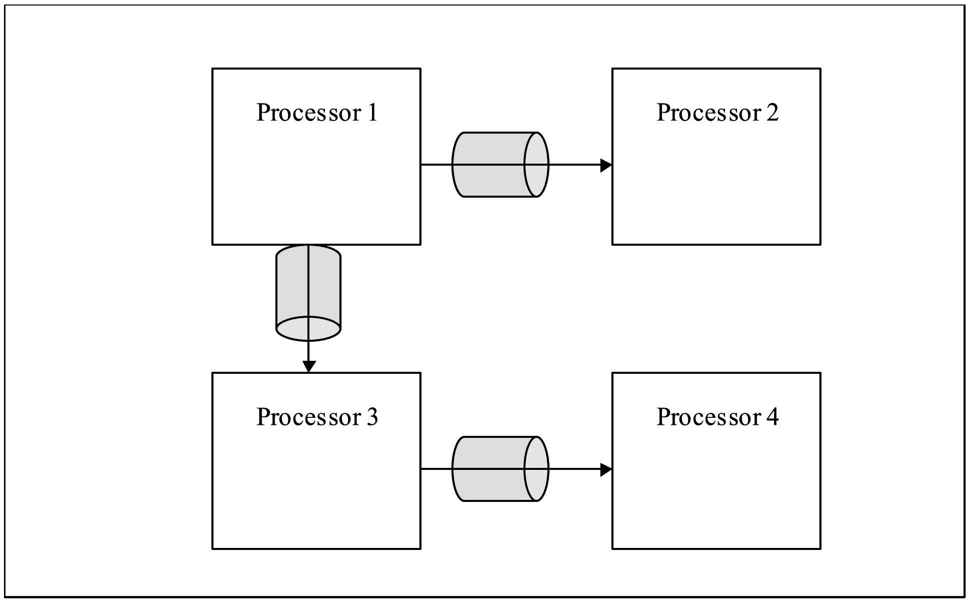


Figure 19: Hardware Resource Model

The class of hardware platforms that we can model with a Resource Model is representative of actual DSP systems used in industry. One such example is a modular hardware platform specified by Texas Instruments that uses DSP processors as the processing nodes of the system. The DSP processors used (TI TMS320C4x and TMS320C6711) are designed for flexible, efficient, low-overhead point-to-point communication between processors. Each processor has a number (typically 4 or 6) of communication ports. Each comport is driven by its own, independent DMA co-processor, consuming no DSP processor cycles. This results in a high degree of concurrent processing and communication.

Real-Time Dataflow

A dataflow semantics, Real-Time Dataflow (RTDF), has been developed specifically for the modeling of real-time systems. Unfortunately, not all dataflow systems can be implemented with the static scheduling scheme used with the Berkeley SDF computation model [50]. Many systems have computations whose behavior varies dynamically at runtime. These variations can be a result of computation behavior that is data-dependent. The dataflow semantics presented in this section retain the same basic semantics as Dataflow Process Networks (reviewed in Chapter III). Two of the most important modifications made to the dataflow semantics presented in this section are the additions of task execution time information and non-deterministic outputs.

We need to explore quantitative properties, since the systems of interest are real-time systems. To accomplish this, we will need to add timing information to our RTDF model. We will now require an execution time interval to be specified for each dataflow computation. This model incorporates the notion of time where each computation block in the dataflow model has some duration. This duration may be modeled as a time interval that represents a range of possible computation times the dataflow component may take to perform the computation. This feature may be used to capture either 1) inherent non-deterministic characteristics in the hardware the component is implemented (e.g. cache affects) or 2) non-determinism in the computation itself such as a data-dependent execution. It is necessary to model these non-deterministic situations since the level we are modeling our system at (dataflow computations) does not model such things as: instruction level execution of computations, memory caching, or any execution within a dataflow component.

To model uncertainty in the behavior of the dataflow tasks the concept of non-deterministic output ports was added. Each output port can be labeled as 1) always (unconditionally) producing a token on this port each time the task executes or 2) sometimes (conditionally) producing a token on this output port. The non-deterministic behavior in the analysis may be due to lack of information in the model (i.e. modeling computations as black-boxes, ignoring internal data-dependent operations). For example, this analysis only captures the system behavior at the dataflow token level. Since the internal behavior of the tasks/computations is not modeled, the task/computation may have data dependent behavior.

An RTDF model is defined by $R = \langle N, F \rangle$, contains a set of nodes $n \in N$, a set of *token flows* (directed edges) $f \in F$, where $F \subseteq (N \times N)$. The nodes (N), and the data-flows (F) form a directed graph where the computations are the vertices and the data-flows are directed edges representing the flow of data tokens between computations. Nodes are defined to be one of the following types: computations (C), sources (S_O), and sinks (S_k).

Token Flows. All token flows have an implied buffer that provides the queuing between computations. This mapping from token flows to buffers is captured by $\boldsymbol{f_B}$, where $f_B : F \to B$. Exactly one buffer, B, is associated with each token flow, F. f_{BL} is a data-flow buffer queue *length* assignment function, where $f_{BL} : B \to \mathbf{N}$ associating a non-negative integer with a dataflow arc buffer. The FIFO buffer queue $b \in B$ on each token flow has an initial marking of tokens present in the queue defined by f_M : B→N, where **N** is a non-negative integer less than f_{BL}(b).

Sources. A source represents the production of data tokens from the external environment. Source nodes have an associated period. Source nodes also have an associated phase parameter [0,P).

Sinks. A sink is a data consumption node. It may have input ports. It does not have output ports.

<u>Computations – input and output ports:</u> Each component has an associated set of input ports $\{I_P\}$ and output ports $\{O_P\}$ that the destinations and sources of token flow arcs connect, respectively. Each output port is defined (labeled) to be a conditional (COND) or unconditional (UNCOND) output. Unconditional outputs always output a token upon completion of computation execution. Conditional outputs may or may not output a token at this time.

<u>Computations – firing rule:</u> Computations are only executed, or *fired*, if their firing rule is satisfied by the tokens available on its input ports and if buffer space is available on its output ports. When a computation is fired, it consumes data tokens on a set/subset of its input ports and produces data tokens on some set/subset of its output ports. Each computation has an associated firing rule, λ. The firing rule defines the data that must be available on a computation's input ports. λ may be IF_ANY, IF_ALL, or user defined, arbitrarily complex, firing rule (currently not supported). The IF_ANY firing rule will cause the computation to be fired if data is present on any of the computation's input ports. The IF_ALL firing rule will enable the firing of a computation if data is present on all of the computations input ports.

Computations – time interval: Each computation, C, has an associated time interval T that defines a discrete range of integer values. The time interval T is specified as (t_L, t_U), where t_L is the lower bound execution time and t_U is the upper bound execution time. Each of these values in the interval T is a possible execution time of the computation. This execution time interval is assumed to be associated with the computation/task executing on the processor to which computation is assigned.

Computations – priority assignment: f_p is a *priority* mapping function $f_p : C \rightarrow N$ is defined associating a non-negative integer, N, with each computation C. This priority is local to a processing element, where P is a priority specified relative to other tasks allocated to the same hardware resource. Multiple computations allocated to the same hardware resource may have the same priority.

Computations – processing resource assignment: A mapping function, f_R (C), is defined to map each component to a hardware resource. Each f_R (C) returns an $f_R : (C \rightarrow R)$

Token Flows – processing resource assignment: f_{FA}: $F_F \rightarrow L$ is a hardware resource mapping function that maps token flow arcs to communication links in the resource model. This mapping is only valid for the set of token flows arc that "flows" across a communication link, F_F. If the source and destination computation of a token flow are not mapped to the same processor then that link belongs to the set F_F of inter-processor token flows, where $F_F = \{ f \mid f_R(c_1) \neq f_R(c_2), where\ f = (c_1, c_2) \}$.

The RTDF model is classified as a homogeneous dataflow model. If a dataflow model is homogeneous then its computations may only input at most one token per invocation on each of its inputs and may only output at most one token on each of its outputs [44].

Real-Time Dataflow Realization on System Resources

The RTDF dataflow computation model is implemented on a multi-processor network described by the hardware Resource Model. A design and execution environment supporting system synthesis has been created at ISIS. It is called the Adaptive Computing System Model Integrated Development Environment (ACS-MIDE) [51]. The ACS-MIDE is an infrastructure consisting of 1) a modeling tool to capture the system dataflow and resource models, 2) a system synthesis tool that interprets the system models and produces information needed for the implementation of the dataflow computation on the processor network, and 3) a runtime environment consisting of a lightweight microkernel (ACS-kernel) that runs on each of the processors in the multi-processor network [52]. The kernel facilitates the execution of the dataflow computation on the multi-processor network. RTDF dataflow computations become kernel *tasks*; a kernel *stream* manages the flow of data between kernel tasks.

The configuration of the runtime environment is a target for system synthesis tools, not configuration by the user. The synthesis tool generates information necessary to produce: 1) a task schedule for each processor, 2) task communication information (information for the creation of streams), and 3) information needed for the bootloader (the bootloader is responsible for loading the proper object code onto each processor in the system). Each processor runs its own copy of the kernel. The kernel is responsible for three main system management functions: scheduling, communication, and memory management. Ultimately, the goal is to construct a finite-state behavior model representing a system configuration (instantiation) that may be synthesized by the ACS-MIDE environment. Towards this goal, the behavior of a RTDF graph distributed across a multi-node hardware platform (defined by the Resource Model) must be defined. System implementation detail is given in the following sections in order to understand the operation of an ACS-MIDE generated system. The two areas we are concerned with that affecting system (timing) behavior are the processor task scheduling and task communication functions (memory management does not influence system behavior for the level of abstraction we have chosen to model our system).

Kernel Task Execution Policy

In an ideal dataflow model, the dataflow computations may execute concurrently if their firing conditions are satisfied. Necessarily, there are further restrictions on task execution when the model is implemented on actual hardware. An RTDF model may be implemented on a single processing node. In this case, the execution of the dataflow computations (in the from of kernel tasks) must be serialized for execution on the processor. An RTDF model may also be implemented on a hardware platform with more than one processing node. Computation nodes of the RTDF graph are allocated to the various processing nodes of the system. In this case, each node may be executing one task.

The kernel implements a priority-based non-preemptive dynamic dataflow scheduler. Dynamic scheduling does incur some overhead cost, but is necessary to implement the dataflow semantics described above. Other scheduling schemes could also be used, but priority-based has an advantage in that tasks may be favored for execution over others. This can be useful in many situations. For example, we may want a task that sends its output data to a task that is allocated on another processor to execute as soon as possible so that the other processor may be better utilized.

The RTDF semantics allow a task to be executed when its input conditions are satisfied and it has buffer space available on each of its outputs. Each time the kernel dynamically schedules the next task to execute, it evaluates the current "state" of the dataflow graph (a portion of the system dataflow graph) on its processor. The state of a dataflow graph consists of the number of tokens present in each dataflow arc buffer. If no tasks are available for execution, then the kernel on that processor simply waits for a change in system state due to incoming tokens from other processors. If more than one task is eligible for execution, then the highest priority task is executed. If the multiple tasks with identical priority are eligible for execution and their priority is the highest priority of all eligible tasks, then the order of execution of these tasks is not specified.

When a task begins execution, the triggering tokens present on the task's inputs are removed from its stream (dataflow token arc) buffer so that the task may operate on them. After a task completes its execution, the output tokens produced by the task are placed in their corresponding dataflow arc buffers on the tasks outputs.

Kernel Task Communication

The kernel seamlessly manages the transfer of data packets (tokens) from one task to another. The mapping (allocation) of an RTDF model onto the system hardware (modeled by the Resource Model) determines what dataflow token arcs in the RTDF model flow across the physical communication link connection between processors. This allocation influences the execution timing of the RTDF model. If both tasks are allocated on the same processor then the data transfer is implemented as a memory copy operation (or, more efficiently, a pointer copy). If the communicating tasks are on different processors the data packet must be transferred between processors via some physical transmission medium (a cable). The on-processor task communication time is negligible; communication between processors is not and must be modeled.

Another difference between on-processor and inter-processor task communication is that there is no "flow-control" for inter-processor task communication. Specifically, sending processor has no knowledge of the state of the communication hardware such as the number of data packets present in the communication hardware FIFO queue. If the sender sends a data packet to communication hardware that has a full FIFO queue, then that data packet will be discarded. The RTDF dataflow semantics are violated and program execution is incorrect.

Generation of Finite-State System Representation

In order to use model checking technology, we must first transform our system model into a representation that is amenable to this type of analysis. This section describes the mapping/semantic translation of the model of an RTDF-modeled system into a behavior model. Any bounded, synchronous system with bounded resources can be modeled by a finite-state model [53]. A timed model, TRSchart, (introduced in Chapter III) is chosen for the representation of the system behavior model. The statechart-like TRSchart model is used to provide hierarchy and concurrency features needed along with a timed semantics necessary to capture timing behavior of the system. The use of a timed semantics allows for modeling of system behavior such as data input rates and task execution times. It also provides the opportunity for analysis of temporal properties of the system (performance analysis). The finite-state model should represent the system behavior as accurately as possible while abstracting away details that do not have any impact on the behavior (timing behavior) of the system.

The ability to represent concurrency is necessary because our system has major components that operate largely independent of each other (with some communication between them). The major system components are: data sources, processing elements, communication links. The orthogonal modeling of these components provides a natural partitioning of the system state space. Consequently, the behavior model of each instance of these system component types may be dealt with independently. The TRSchart behavior model will have the following structure: The root state will be an OR state. A child of the root state will be an AND state labeled “System”. The “System” state will have a child OR state for each major component of the system. Each of the component OR states will contain the behavior model of that component. A TRSchart system model with this structure is shown in Figure 21.

From this model we see that the global “state” of the system is comprised of the state of each of the primary system components: processors, communication links, and source generators. Sink nodes are not active components but simply a placeholder to mark a system output; as such no behavior model is needed. A sub-TRSchart behavior model of each component is independently constructed. Interactions between the system components occur only via shared events and guarding state conditions (guarding conditions allow a module to have conditional behavior based on the state of another module). Figure 20 and Figure 21 show an example RTDF model implementation and its corresponding TRSchart behavior model, respectively.

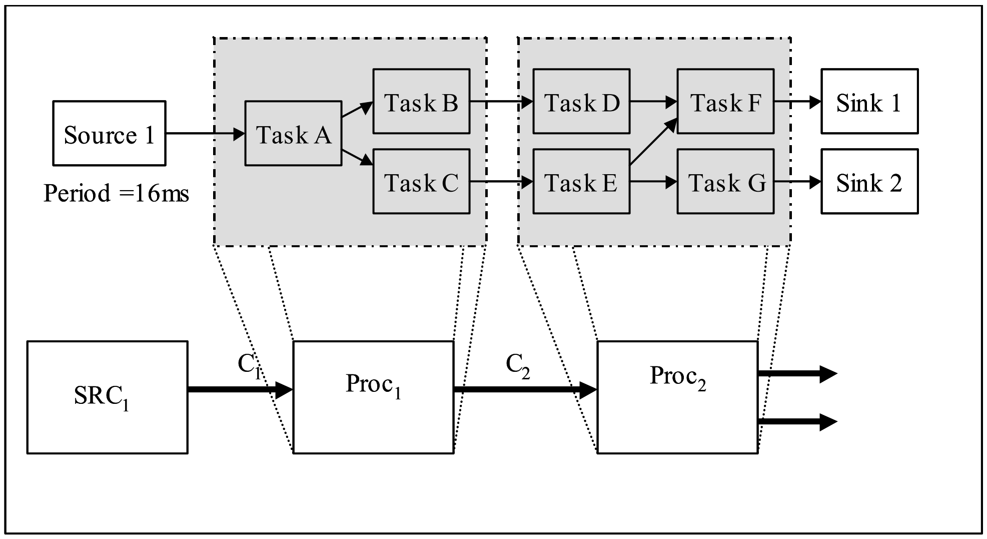


Figure 20: RTDF program implemented on two-processor network

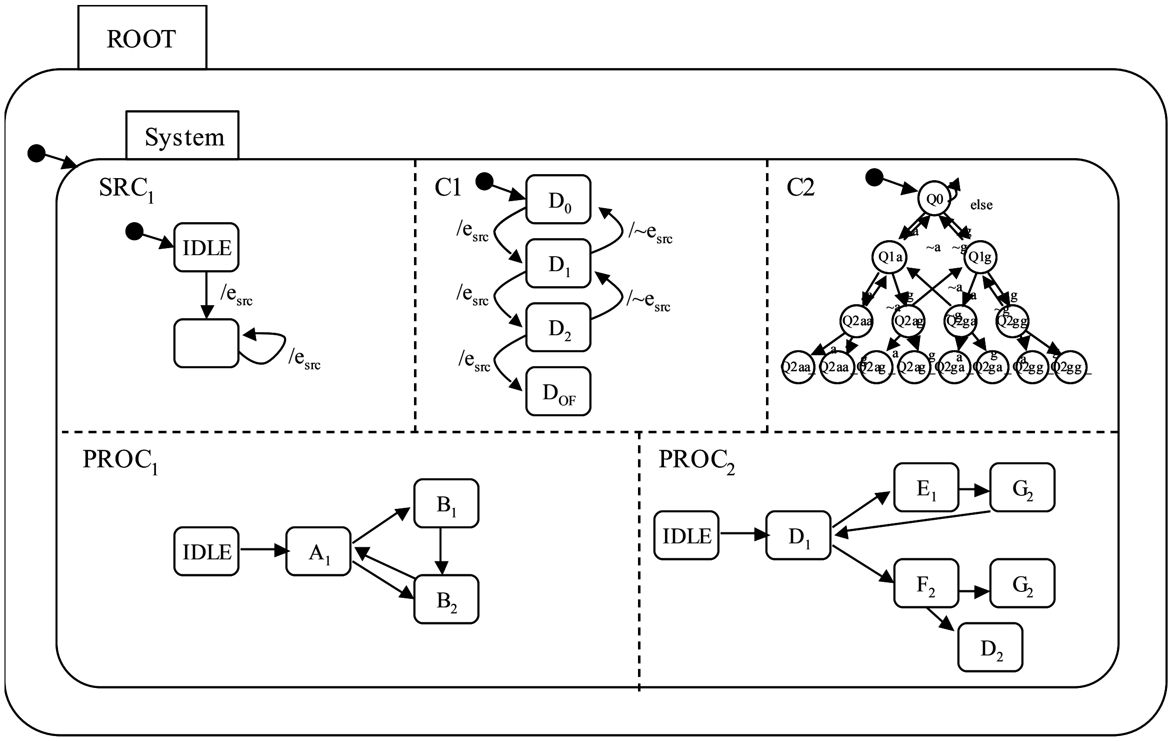


Figure 21: TRSchart finite-state behavioral model of system in Figure 20

The following sections describe the mapping from our system model to the generation of a system behavior model composed of behavior models of each of the primary system components: Processor, Communication Links, and Token Sources.

Generation of Communication Link Finite-State Behavior

A TRSchart communication link behavior graph is created for each physical communication link in the system Resource Model. A commlink provides both the physical transport for data packets (tokens) and buffering in the form of an ordered FIFO queue. A commlink has two main attributes needed for the construction of its behavior model: buffer depth, L_D, and the set of token flow arcs that this link must service (i.e. the number of flows multiplexed across this link), defined as L_F. The "state" of a commlink captures the contents of its ordered fifo queue, retaining knowledge for each token in the queue as to which token flow arc it belongs. Therefore, the number of states needed to model a commlink is:

$$\sum_{i=1}^{L_D}\left(|L_F|^{i}\right)+1$$

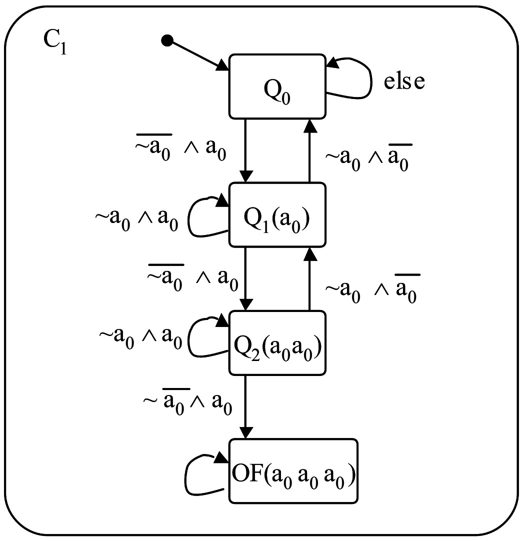
The commlink model should also trap overflow conditions if they occur. When an enqueue is attempted on a full commlink the model should transition to an exception state. Although this could be accomplished by adding a *single* exception state that the model transitions into when an enqueue occurs at any "full queue" state, instead we chose to put a separate exception state for each unique overflow condition. This will provide more information to the verification tool as to the state of the queue when an enqueue event triggered the overflow. The following total number of states needed to model a commlink is updated to include the extra "layer" of exception states:

$$\sum_{i=1}^{L_D+1}\left(\left|L_F\right|^{i}\right)+1$$

The commlink behavior model uses events from other concurrently active behavior models to signal the enqueuing and dequeuing of data. At most one enqueue and one dequeue event can occur at any time because the communication only services one source processor and one destination processor. An enqueue and dequeue event may occur simultaneously. Each of the operations (enqueue, dequeue, and simultaneous enqueue and dequeue) that the behavioral model must handle from any state is described below.

Enqueue: At any time, a commlink may receive an enqueue event from a computation on an upstream source processor. This input may be from any of the $f \in L_F$ token flows mapped to, or multiplexed on, this commlink. Each state in this behavior model must react to each of the of the $|L_F|$ possible enqueue events (except overflow exception states). Transitions are created to implement this behavior. For example, the transition in Figure 22 from state "$Q_1(a_0)$" to state "$Q_2(a_0a_0)$" shows an enqueue operation. The firing condition of this transition is labeled "$\neg \sim a_0 \wedge a_0$". Event 'a_0' is the enqueue event signaled from the upstream processor, labeled with the name of the computation (or token generator) that produced the token and an index of the token flow from the computation ensure label uniqueness. This is important for the case where computation produces tokens from multiple outputs that will be transmitted across the same communication link. Dequeue events use the same labeling as their corresponding enqueue events, but with a '~' attached to the beginning of the event label. Event '$\sim a_0$' represents a dequeue signal from the downstream processor. From this we see that the transition represents an enqueue operation if an enqueue event is present and a dequeue event is not present.

Dequeue: At each state the commlink should react to the $|L_F|$ possible dequeue events. The commlink will remove the next token to be sent out from its queue and transition to the state reflecting its new queue contents. Refer back to Figure 22 to the transition from state "$Q_2(a_0a_0)$" to state "$Q_1(a_0)$" for an example. This transition is labeled "$\sim a_0 \wedge \neg a_0$". The transition implements the behavior: if dequeue event '$\sim a_0$' is present and enqueue event 'a_0' is not present, then perform a dequeue operation.



C_1
Q_0
else
$\overline{\sim a_0} \wedge a_0$
$\sim a_0 \wedge \overline{a_0}$
$\sim a_0 \wedge a_0$
$Q_1(a_0)$
$\overline{\sim a_0} \wedge a_0$
$\sim a_0 \wedge \overline{a_0}$
$\sim a_0 \wedge a_0$
$Q_2(a_0a_0)$
$\sim \overline{a_0} \wedge a_0$
$OF(a_0\,a_0\,a_0)$

Figure 22: Communication channel behavioral model with $L_F=\{a_0\}$, $L_D=2$

Simultaneous enqueue and dequeue: The behavior model must also capture the behavior of the case that both an enqueue and dequeue event occur simultaneously. A behavioral model is shown in Figure 23 for $L_F=\{a_0, b_0\}$, $L_D=2$. The state labeling convention in Figure 23 is that the most recent token placed in the queue be labeled as the rightmost portion of the name and the rest of the queue contents be listed right to left, in order of arrival.

Consider simultaneous enqueue of a token from some token flow f_1 and a dequeue event from some token flow f_2. The next state of the system is determined by what the queue contents will be after the enqueue of a token from f_1 and a dequeue of the next available token. The behavior of a FIFO queue dictates that the first token put into the queue will be the first to be removed. Note that the dequeue event from flow f_2 should correspond with next token that is available to be removed from the queue. This is enforced by actions of the processor behavior model that is consuming the data. The next state should be the queue state with the token corresponding to token flow f_2 removed from the queue and a token corresponding to token flow f_1 inserted in the most recent (first in) queue position. For example if we are in state "$Q_2(a_0a_0)$" and we receive both an enqueue event, b_0, and a dequeue event $\sim a_0$, then the behavior model should transition to state "$Q_2(a_0b_0)$". This is shown in Figure 23.

Overflow Exception: The FSM behavior model of the communication link captures the case that the channel queue may exceed its capacity. Once the FSM model transitions into an "overflow" state it will remain in this state forever. There are no outgoing transitions from the overflow state – except for the implied loopback transition. Since this is a critical system error condition, if it ever occurs execution of the behavior model of the communication link is frozen. Independent states store FIFO queue contents when an overflow occurs.

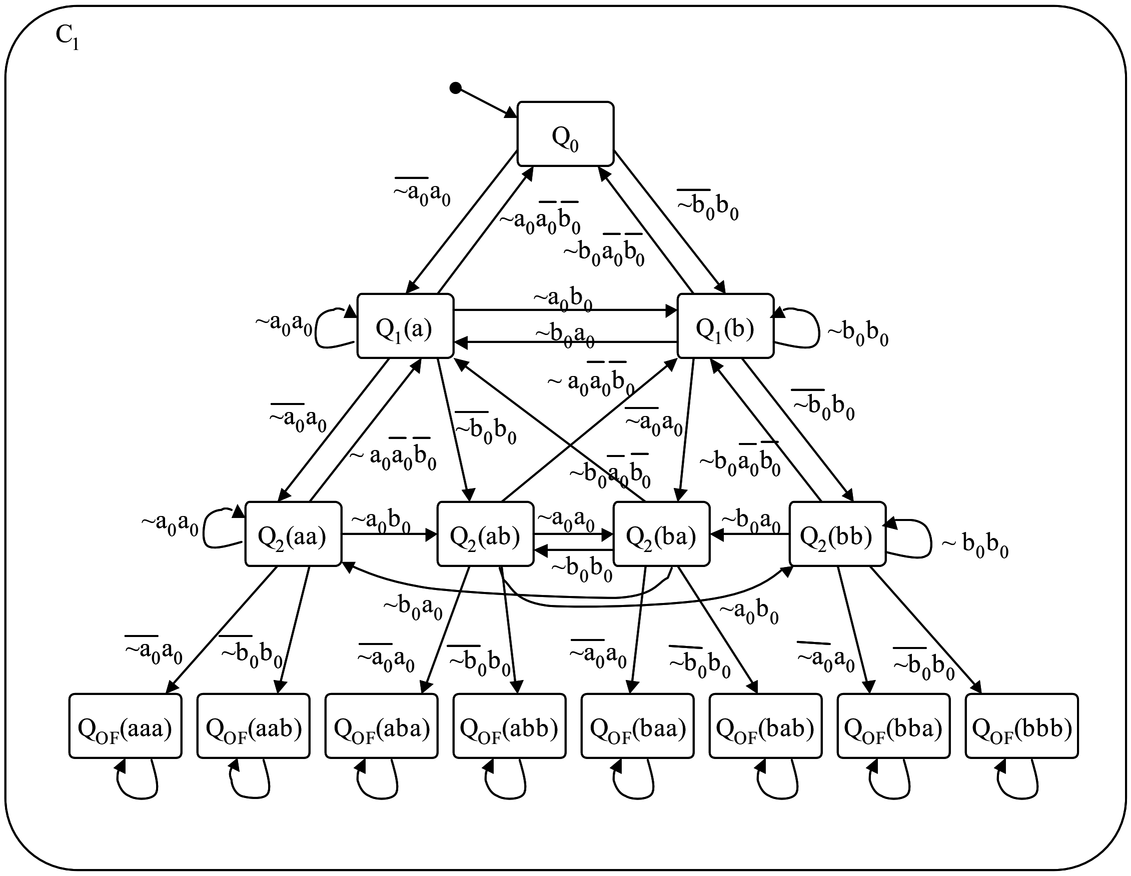


Figure 23: Communication link behavioral model, $L_F = \{a_0, b_0\}$, $L_D=2$

Generation of Processor Finite-State Behavior

A finite-state behavior model (TRSchart) is constructed for each processor in the system. This model should capture all possible behaviors of the system implementation as occurring on this processor. The processor has some portion of an RTDF model that it is responsible for executing. First, we define what is "state" of our finite-state behavior model of the processor.

Definition of Processor System State

The “state” of an SDF dataflow model is defined by the number of tokens queued on the buffer on each dataflow arc [54]. Similarly, to capture the state of a processor node of the system, we must capture the state of the “local” dataflow arcs of the portion of the RTDF model mapped onto a processor P. A dataflow arc is local to processor P if both its source and destination tasks are allocated on processor P. Also needed to capture the state of a processor is the currently executing computation. In this way we have captured the state of the processor as a “snapshot” at the point where a computation is currently executing. The information needed to capture the state of a processor *P* in the behavior model is summarized as follows:

Let *PS* be a tuple $\langle c_E, f_{BS} \rangle$, where:

c_E is the currently executing computation, where $c_E \in C_P$. C_P is the set of all computations on a processor P, where $C_P \subseteq C$.

$f_{BS}(b)$: $B_P \rightarrow N$, where with $0 < N < f_{BL}(b)$. f_{BS} is a mapping associating a number of occupied slots of a buffer to each $b \in B_P$. B_P is the set all buffers on all local flows, F_P, where $B_P = f_B(F_P)$ and $F_P = \{ f \mid f_R(c_1) = f_R(c_2), where\ f = (c_1, c_2) \}$

The processor is said to be *idle* ($c_E = \varnothing$) if there is not a task available for execution. The maximum number of states of a processor behavior model will be the following:

$$Max\_States = |C_P| \times \prod_{\forall b \in B_P} f_{BS}(b)$$

Processor Behavior Generation

We now know what constitutes the "state" of the processor and we are ready to begin the construction of a finite-state behavior graph. A separate behavior graph is built independently for each processor instance present in the model. Since we now have defined discrete states of the processor, we can produce a next-state function, *FindNextProcStates*, defining the behavior of the processor.

The following describes how the *FindNextProcStates* function evolves a processor state ("procstate" for short), ps_1, into a set of *all possible* next-procstates, $PS_{Next} \subseteq PS$.

<u>Step 1</u>: The transformation to the next-state of a processor begins at the point where a computation c_E has finished execution. The output buffers of c_E are updated to reflect its completion by producing tokens on appropriate computation outputs. Let B_{UC} be the set of all buffers associated with unconditional outputs of c_E, and B_C be the set of all buffers associated with conditional outputs of c_E. Increment the value of the set of buffers B_{UC}.

As defined by the RTDF model, conditional outputs may or may not produce tokens. The next-state function must consider each of these cases as a possible non-deterministic behavior. For each member of $B_{CN} = 2^{B_C}$ we will create a new procstate, ps, where the number of occupied slots of the buffers of each member set of B_{CN} are incremented. The set of all procstates, ps_i, are assigned to a set PS_{Next}.

<u>Step 2</u>: Each processor must react to input data tokens that become available on its communication links. At each step in the simulation, all possible reactions to data available from sources external to the processor on incoming commlinks are considered. Let B_{EP} be the set of all buffers associated with the external inputs to this processor. For all $ps \in PS_{Next}$, we create a set of procstates that represent the possibility that any possible set of incoming data could arrive in the set of buffers, B_{EP}. Let $B_{IN} = 2^{B_{EP}}$. For each ps$\in$ PS_{Next} we will replace each ps element with a new set of $|B_{IN}|$ elements where each of these new ps elements is modified to increment the value of its b $\in B_{IN}$ buffer set.

<u>Step 3</u>: A computation is eligible for execution if it has the highest (or tie for highest) priority of all computations on this processor that have their firing conditions met. Therefore, it is possible that the highest priority computation eligible for execution is not a single computation, but a set of computations each having the same priority. Let C_{NEXT} be the set of computations eligible for execution at the next scheduling period, where $C_{NEXT} \in C_P$. If multiple computations are eligible for execution, the set of possible next-states, PS_{Next}, diverges again into | PS_{Next} | x $|C_{NEXT}|$ behaviors for each of the $|C_{NEXT}|$ possible behaviors.

Since the order of the executions of these tasks is unspecified, all possible orderings are considered in a non-deterministic manner. For $|C_{NEXT}|$ computations with the same priority, the execution diverges into $|C_{NEXT}|$ execution traces to model the case where each of the $|C_{NEXT}|$ computations may be selected for execution at this time.

For all ps$\in$ PS_{Next}, each ps is replaced with a set of $|C_{NEXT}|$ procstates where a new procstate is created from the original ps with each of the new procstates takes on a c_E assigned a member of C_{NEXT}.

<u>Step 4</u>: At this point each potential next-procstate ps$\in$ PS_{Next} has selected the new computation, c_E, it will execute. We would now like to update all next-procstates to reflect the tokens that the computation, c_E, will consume.

For each ps$\in$ PS_{Next}, decrement all input buffers of task c_E in the following manner: if c_E has an IF_ALL firing rule, then decrement the number of occupied slots of all buffers associated with incoming token flows of c_E; if c_E has an IF_ANY firing rule, then decrement the number of occupied slots of the buffer associated with the incoming token flow that first triggered the firing of c_E.

In summary, from a procstate we find the set of all possible next-procstates to which the system may transition. From the next-procstate description above we can see that the number of potential next-procstates a system may transition to is: $2^{|B_C|} + 2^{|B_{EP}|} + |C_{NEXT}|$.

Building the behavior graph is a straightforward process. The TRSchart model behavior graph is built up incrementally as we traverse the processor's behavior via the *FindNextProcStates* function. We begin by considering that the processor is in an initial *IDLE* state where the processor has no currently executing task and all local dataflow arc buffers are set to their initial marking as defined in the system model.

$$ps_{IDLE} = \{c_E = \emptyset, f_{BS} = f_M\}$$

A TRSchart state is created representing the ps_{IDLE} procstate and is marked as the initial state in the behavior model. Using the *FindNextProcStates* function the set of all potential next-procstates is found, PS_{Next}. A corresponding TRSchart state is created for each $ps \in PS_{Next}$. For each TRSchart state created, a time duration interval is assigned to it corresponding to the execution time interval of the currently executing computation, c_E, of the procstate ps this state is to represent. When we create a TRSchart state to represent a procstate, we first check the TRSchart to see if that state exists. If that state exists, then we know that procstate has previously been encountered then we do not traverse that path further. To facilitate this, an encoding function is used that creates a unique bit encoding for any procstate $ps \in PS$. The exploration continues in a breadth-first manner. The process terminates when there are no new states to explore. Given each processor model has a finite number of states, the construction of the behavior model is guaranteed to terminate.

A transition is created for each of the possible behaviors leading from the state representing the original procstate to each of the states representing its possible next procstates. Transitions representing processor behaviors that interact with the behavior models of other system components must be labeled with the proper firing conditions to signal the input of tokens from an external commlinks and the output of tokens to external commlinks, where appropriate.

If the current procstate produces an output(s) destined for a computation(s) on another processor (i.e. c_E outputs to an inter-processor token flow, $f \in F_F$) then all transitions leading from this procstate must send an *enqueue* output event(s). This event signals an enqueue to the appropriate commlink model. This output event is assigned a label composed of the computation name of c_E and an index assigned to the outgoing token flow f (this guarantees a unique event label). This event will signal an enqueue event of a token from a particular flow to a concurrently active commlink behavior model.

If a transition between procstates represents the behavior of a token being input from a communication link queue, then this transition should only be allowed to fire depending on the state of a concurrently operating communication link behavioral model. This transition will be labeled with a guarding condition specifying that the transition only occur if data destined for this buffer is the next available token in the commlink FIFO queue. Upon firing, this transition should also send a *dequeue* event signal to the commlink model to acknowledge that we have accepted a token from the communication link queue. The dequeue output event is labeled the same as its corresponding enqueue event, but with a '~' added to the beginning of the event label. For example, refer to the system model in Figure 24, corresponding processor behavior graph segment in Figure 25 and corresponding communication link behavior graph in Figure 22. The transition from S_1 to S_3 only occurs if the commlink has a token available from its FIFO queue to flow f_1. In terms of the behavioral model the queue must be in state "$Q_1(a_0)$" or state "$Q_2(a_0a_0)$" for the transition to fire. If the conditions are met for the transition to fire, then a dequeue event should be produced. A transition label is created for this transition to represent this behavior, $[Q_1(a_0) \vee Q_2(a_0a_0)]$ /$\sim a_0$.

There is a subtlety that must be addressed. When a dequeue event is sent to a commlink, it takes a clock cycle for it to react to the event and change state indicating that the token has been removed from the queue. During that clock cycle, the processor behavior model may react to the current state of the commlink that does not yet reflect the dequeue operation. This can cause the processor behavior model to incorrectly dequeue the same token from a commlink FIFO twice (although only if a computation has a lower bound execution time of one). Another triggering event must be added to the transition label that inhibits the firing of a transition if the dequeue event is still pending. In Figure 25 we can see the expression "$\neg\sim a_0$" is added to the transition label to produce: $[Q_1(a_0) \vee Q_2(a_0a_0)] \wedge \neg\sim a_0$ /$\sim a_0$.

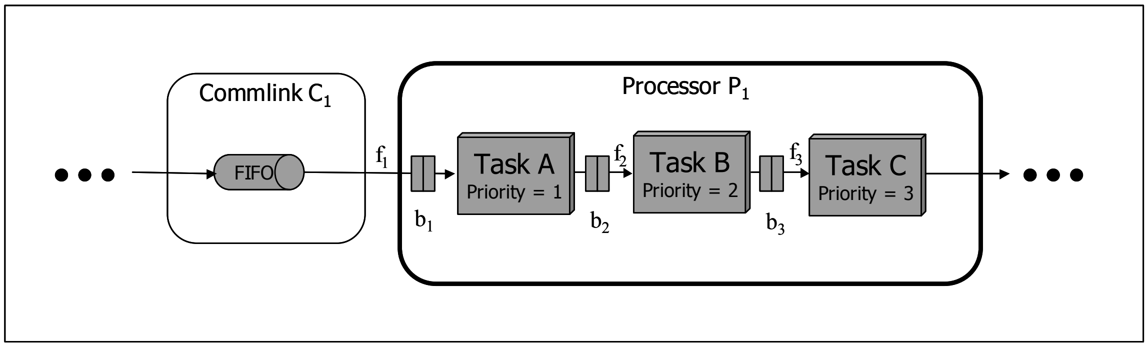


Figure 24: Example RTDF model mapped onto hardware

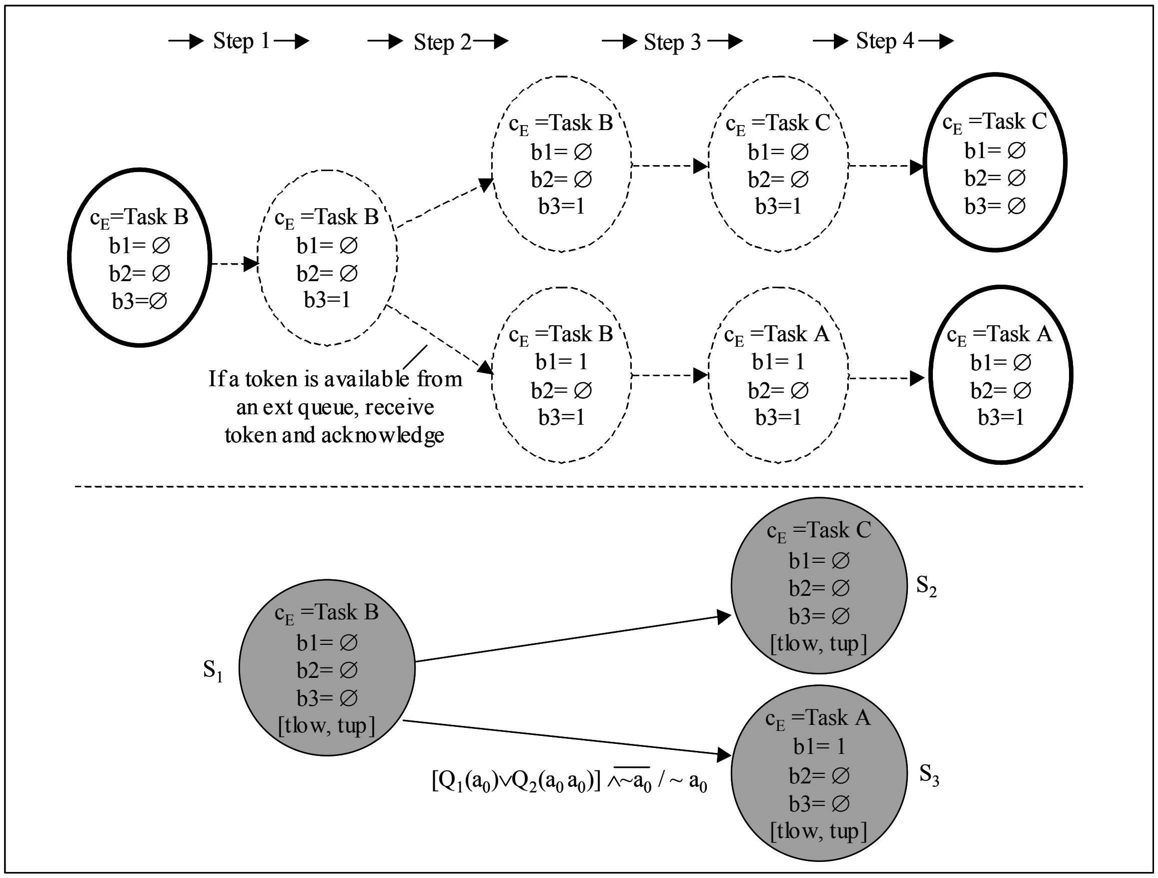

Step 1
Step 2
Step 3
Step 4
c_E=Task B b1=∅ b2=∅ b3=∅
c_E =Task B b1=∅ b2=∅ b3=1
c_E =Task B b1=∅ b2=∅ b3=1
c_E =Task C b1=∅ b2=∅ b3=1
c_E =Task C b1=∅ b2=∅ b3=∅
c_E =Task B b1= 1 b2=∅ b3=1
c_E =Task A b1= 1 b2=∅ b3=1
c_E =Task A b1=∅ b2=∅ b3=1
If a token is available from an ext queue, receive token and acknowledge
S_1
c_E =Task B b1=∅ b2=∅ b3=∅ [tlow, tup]
S_2
c_E =Task C b1=∅ b2=∅ b3=∅ [tlow, tup]
S_3
c_E =Task A b1= 1 b2=∅ b3=∅ [tlow, tup]
$[Q_1(a_0) \vee Q_2(a_0\, a_0)]\ \overline{\wedge \sim a_0}\ /\sim a_0$

Figure 25: Processor behavior model generation example

Generation of Data Source Finite-State Behavior

All systems will have one or more continuous streams of data as inputs to the system. Each of these data sources is modeled as *token generators*. A TRSchart behavior model is created for each token generator. Each model simply contains the period, $t_{period,}$ and phase delay, t_{pdelay}, of each data source. An example of the behavior TRSchart behavior model of a source is shown in Figure 26. Only two states are necessary. The first token is output t_{pdelay} time units after entering the IDLE state (system start). All subsequent tokens are output at t_{period} units of time.

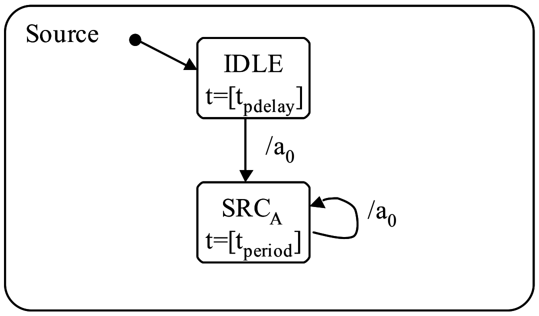


Figure 26: Periodic Source Generator

System Verification

For the problem domain of data-driven multi-processor real-time systems, the primary focus is to verify that the system will operate correctly. In this work, “the system” is specified by a RTDF model implemented on a hardware platform defined by the Resource Model. Because the system is a *real-time* system, timing information must be modeled. As detailed in the previous section, a timed model, TRSchart, is chosen to model the behavior of the system.

The primary goal is to verify that the system is *schedulable* for all possible system behaviors. We define schedulability as follows: *"If a system is able to execute its dataflow graph continuously in all possible cases, then the system is schedulable."*

The asynchronous nature of the dataflow model causes tokens to flow throughout the system at a self-regulated rate. If processing "gets behind" relative to the incoming data rates, then data buffers throughout the system will consequently begin to fill. Data flows local to a processor maintain lossless flow control; data flows that span across processors do not maintain flow control. The hardware communication FIFO queues are lossy. That is, there is no "data ready" sent back from receiver to sender indicating that there is a free slot in the queue for a data token to be stored. Therefore, the sender has no knowledge of the queue state and may send a data token to a communication link with a full queue. When a data token is sent to a communication link with a full FIFO queue, the data has nowhere to go and is discarded. As shown earlier in this chapter, the behavioral model captures this behavior by entering an *exception state* if this condition occurs.

Schedulability of the system model can be proven to hold true by checking that our inter-processor communication link buffers do not overflow for *any* possible system behaviors. A Model checking technique, reachability analysis, can be used to prove that the system behavior model never exhibits certain behaviors, specifically the token loss as described above.

Reachability Analysis

System verification problems can often be cast into a *reachability analysis* problem. Reachability analysis of a finite-state machine is the computation of all reachable states from an initial state (or state set). Reachability analysis can be used to prove whether or not it is possible for a system model to enter into an undesirable state. To find the set of all reachable states in a conventional manner, we must visit and mark each state that has been visited. In the case of a TRSchart/RSchart model this requires the storage of all possible system *global configurations* that may be reached from an initial state. Due to the extremely large set of global configurations that a TRSchart may define, this can quickly become unmanageable.

The use of a symbolic representation of the TRSchart/RSchart model can allow a more efficient representation of the large state sets involved. Through symbolic manipulation of the state set representations, analysis of systems with very large state spaces can become possible. Reachability analysis is the core operation for many OBDD-based verifications. The OBDD representation provides efficient set-based breadth first search traversal.

```
Reachability ( State set Init )
{
        $S_2$ = Init;
        do {
                $S_1 = S_2$;
                $S_2$ = ForwardStep ( $S_1$ ) $\cup$ $S_2$;
        } while ( $S_2 \neq S_1$ )
        return $S_2$;
}
```

Algorithm 8: Basic Reachability Algorithm

The basic reachability algorithm begins with an initial state set, and traverses the state space in a breadth-first manner collecting all encountered states into a state set, S_2. The computation of next-states (ForwardStep) is performed on the entire state set, S_2, on each iteration. The algorithm terminates when the next-state set does not return any new states and a fixpoint is reached.

Another version of the reachability algorithm is known as reachability via frontier set traversal, presented in [23]. This algorithm is similar to the previous except the forward step computation is only performed on the set of new states encountered at each breadth-first step through the system state space. This provides a possible reduction in the size of the OBDD that must be operated on in the ForwardStep function. Note that the state set to perform the ForwardStep calculation on can be chosen to be any subset of *Cur_NextStates* ∪ *Reached* that includes the states in *NewStates*, ideally the subset with the smallest OBDD size. As finding a subset with the minimal OBDD-size is in itself a difficult problem, the frontier set approach using *NewStates* to compute ForwardStep is used in this work. As noted in [23], the frontier set approach does little to improve maximum memory usage but does provide a constant-factor speedup in computation time. These algorithms are equally applicable to the MTBDD-based analysis techniques presented in Chapter IV.

```
FrontierSetReachability( State set Init )
{
        Current = Init;
        Do {
                Cur_NextStates = ForwardStep(Current);
                NewStates = Cur_NextStates – Reached;
                Reached = Reached ∪ NewStates;
                Current = NewStates;
        } while ( NewStates ≠ ∅ )
        return Reached;
}
```

Algorithm 9: Reachability Algorithm Using Frontier Set Approach

Schedulability Verification

To prove system schedulability, we now know that it is sufficient to prove that communication link buffers do not overflow. As shown in the construction of the behavior graph, certain system behaviors may trigger system transitions to special exception states that represent overflow conditions. Once the system enters one of these states, it is not allowed to leave. Exception states have no outgoing transitions to states other than itself, only a single implicit transition that has both its source and destination state to itself. *Reachability analysis* is used to exhaustively search the entire state space of our system to prove whether or not it is possible to have a system behavior where a communication buffer will overflow.

One option is to compute the reachable set of the system via Algorithm 8 or Algorithm 9 and check for an intersection of this state set and the set of all exception states (the *exception set*). We, however, are primarily interested in detecting whether or not it is possible to reach an exception state, not necessarily computing the *entire* reachable set. A modified form of the reachability algorithm is used for schedulability verification. As shown in Algorithm 10, the algorithm computes the reachability set as in the frontier set approach of Algorithm 9, but now checks each new state set for an intersection with the exception set. The algorithm terminates when it first detects an intersection between the set of new frontier states and the exception set or a fixpoint is reached. The algorithm returns the intersection of these sets, giving the set of exception states *first* encountered. If no exception states are encountered, then the algorithm terminates when all reachable states are found.

Violations occurring on a breadth-first traversal after the point where an exception set is found are may be due to secondary effects of a previous violation and may not be discerned from first-time violations occurring on another unrelated behavior trajectory. When a communication overflow condition occurs and a communication model enters an exception state, then it will remain in that state. This will obviously cause other violations to occur as the behavior traversal continues.

```
Reachability_ExceptionDet ( State set Init )
{
        $d_{SEQ}$ = 0;
        Current = Init;
        do {
                Cur_NextStates = ForwardStep(Current);
                $d_{SEQ}$++;
                NewStates = Cur_NextStates – Reached;
                If ((NewStates $\cap$ Exception) $\neq \emptyset$)  // an exception state has been encountered
                {
                        return (NewStates $\cap$ Exception);
                }
                Reached = Reached $\cup$ NewStates;
                Current = NewStates;
        } while ( NewStates $\neq \emptyset$ )
        return Reached;
}
```

Algorithm 10: Modified reachability algorithm

As the state space is traversed in a breadth-first manner, from one state set to the next, note that there is no concept of individual “path” or “trajectory”. When it is found that a state or state set may be reached, indicating the system is not schedulable for a particular behavior or behaviors, we would like to find out why. A counterexample should be provided to the user for every distinct behavior that will lead to an exception state.

Counterexample Generation

If the schedulability analysis determines that the system can exhibit faulty behavior, the next question is "what behavior trajectories lead to the faulty behavior?" The designer needs to be able to track down the source of the problem. To assist in this, the verification tool implements a *counterexample* feature.

When a system specification is violated, the tool can return all error traces, or *counterexamples*, that may lead to an exception state. In practice, however, the user is often not interested in counterexamples that lead to *all* the exception states we could possibly reach. If we reach an exception state on a given behavior trajectory, then any further exception states we encounter on this trajectory will very likely be a consequence of the first exception state (the communication link model is inactive after it reaches an exception state). Due to the lack of any path information when performing a state *set* traversal, its cannot be known if subsequent exception states encountered are first violations on other trajectories or simply further exceptions generated on the same paths that previous exceptions were found. It is more informative to return the counterexamples of the paths that lead to the first exception state (or state set), labeled S_E, that are reached via breadth-first traversal of the system.

We want to find all trajectories and conditions, from an initial state, that leads to a state or states that are members of an exception state set. We cannot simply individually traverse all paths of our finite-state behavior model until such states are reached, and it is certainly not feasible to store path information for every trajectory. The number of trajectories can simply be too large. This section presents a method to reduce the number of trajectories that must be enumerated so that it is now feasible to find all paths from the initial state to the exception set, S_E.

Verification is performed via reachability analysis presented in Algorithm 10. The modified reachability analysis stops at the point where an exception state set, S_E, is first encountered. Although we loosely refer to both global configurations and state configurations of our TRSchart model simply as *states* of the model, it should be clarified that the set of *global configurations* containing the exception states, S_E, is returned as S_{EG}. This global configuration also captures the state of other concurrently operating machines in the model along with any events that are present at this instant. Also returned is *sequential depth*, d_{SEQ}, which denotes the number of steps from the initial state at which the exception state was encountered.

We begin with a global configuration set, S_{EG}, and find the set of all global configurations that are reachable from a backwards traversal of the TRSchart system model. The *backwards reachability analysis*, as shown in Algorithm 11 is used to find this global configuration set. The backwards reachability set search terminates when either all reachable global configurations are found (a fixpoint is reached), or a backwards BFS sequential traversal depth of max_depth is reached. The backwards reachability set is referred to as set S_B. This set will contain the global configurations that on the paths we are in search of from S_{INIT} to S_{EG} and also many other global configurations.

```
BackwardsReachability ( State set $S_{INIT}$ , int max_depth)
{
        $d_{SEQ} = 0$;
        $S_2 = S_{INIT}$;
        do {
                $S_1 = S_2$;
                tmp = BackwardStep($S_1$);
                $d_{SEQ}$++;
                $S_2 = tmp \cup S_1$;
        } while ( ($S_1 \neq S_2$ ) $\wedge$ ($d_{SEQ} \leq$ max_depth) )
        return $S_2$;
}
```

Algorithm 11: Backwards Reachability

In a nutshell, with our TRSchart system model and backwards reachability set S_B, we can now begin a forward traversal of the system behavior, using set S_B to *prune* the number of trajectories we will explore. If a global configuration is not a member of S_B, then that global configuration must not be on a path leading to the exception global configuration set, S_{EG}. Therefore, we can remove that global configuration from our traversal set and continue to BFS forward traversal, logging all path trajectory information as we traverse.

Forward traversal of the behavior model begins once again at S_{INIT}. From S_{INIT}, we compute the set of next-states (global configurations) and prune this set by taking its intersection with the backwards reachability set S_B. This operation will prune out (most if not all) trajectories not leading to S_{EG}. The remaining portion of the new global configuration set (represented as a single symbolic function) is now broken down into its member global configurations so that we can traverse behavior paths individually. This process of extracting the members of symbolic state set is described in Chapter IV, "Testing for set membership".

There is no guarantee that all global configurations not present on paths leading to S_{EG} will be pruned immediately. This occurs if a member of S_B that is not on a path leading to S_{EG} is encountered. All paths not leading to S_{EG} will be pruned before S_{EG} is reached.

The paths traversed from S_{INIT} to each of the members of the pruned next state set are recorded. A tree data structure is created with each node of the tree representing a global configuration, and directed connections from one node of the tree to the next represents transitions from one state configuration to the next. In this way, the tree structure records the behavior trajectories that may lead to the exception state. This tree structure can be provided to the user in two formats: a graphical output or a tabular format. The graph format has the advantage of visually showing the tree structure of the path trajectories. The graph format can, however, grow to quickly become difficult for the user to manage both in terms of its size and global configuration state information that needs to be presented at each node. A tabular format is also provided. One file is generated for each counterexample found. Figure 27 shows a summary of the counterexample generation process.

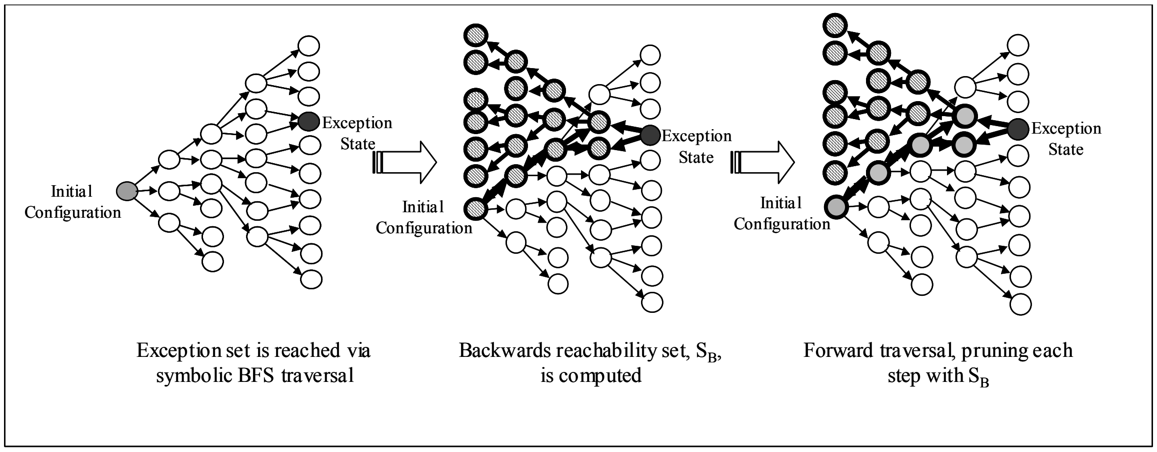

Initial Configuration
Exception State
Exception set is reached via symbolic BFS traversal
Initial Configuration
Exception State
Backwards reachability set, S_B, is computed
Initial Configuration
Exception State
Forward traversal, pruning each step with S_B

Figure 27: Counterexample construction

Quantitative System Analysis

The verification framework has features that can provide quantitative performance analysis information about the system. Due to the nature of the symbolic analysis where only operations on state *sets* are possible, it is only efficient to compute performance *extremes* of the system. A facility has been implemented in the verification tool that will find the maximum system throughput (that is guaranteed schedulable). Currently, this analysis is implemented with respect to a single source node. The search begins with an initial source period, P, entered by the user. The system iteratively performs schedulability verification on the model. If the system is found to be schedulable, then the system model is modified to give the selected source node a new period P = P - N, where N is a period step decrement value supplied by the user. This iteration continues unit the model is found to be not schedulable. At that point the source period of the previous (successful) model is returned as the maximum rate the system can be driven at with respect to that source node ($1/ P_{i-1}$).

Summary

This chapter presented a method for modeling multi-processor embedded systems. A new model of computation, RTDF, was defined. The execution semantics of an RTDF model implementation on a multiprocessor hardware platform was defined. A method of system schedulability verification was presented, including generation of counterexamples when schedulability requirements are violated.

CHAPTER VI

CASE STUDY

Two example systems are analyzed using the verification methods presented in this dissertation. The first is an Automatic Target Recognition system that was built as a demonstration system for ARMY/AMICOM and the second is a health-monitoring system for the Space Shuttle Main Engine.

ATR Algorithm

To evaluate the verification methods presented in this work, an example system has been modeled. This system is an embedded, real-time, automatic target recognition system (ATR) that is implemented on a multi-processor platform. The ATR system implements an image processing algorithm that has the task of finding and classifying all objects in an image. The ATR algorithm is a computationally intense algorithm with a 20 frames/sec image input rate, with each image size being 128×128 pixels and 8-bit depth.

These objects are classified according to a set of four target classes. The algorithm performs a correlation between each of the target classes in the input image. The resulting correlations surfaces are normalized and then searched for local peaks according to a peak-to-sidelobe criteria. A peak-to-sidelobe ratio (PSR) is computed for each local peak. A PSR value above a certain threshold indicates a region where a target object may be found. The top ten PSR values are used to extract a corresponding region of interest (ROI) around the PSR position plucked from the original unscaled image. Each ROI surface is normalized and passed on to their respective distance calculation function. The distance calculation function computes a measure of closeness of the found object to its respective target class. The distance values for each of the four target classes are passed on to a distance compare function whose job it is to merge and order the distances into a list. This merged list of distance measurements is sent to the post processing function where it attempts to further discriminate between targets and finally produces a single target coordinate (in terms of image position).

Algorithm Model

This algorithm has been modeled as an RTDF model as shown in Figure 28. RawImageSrc and HMACH_Filter_Src are Source nodes each with a token generation period of 50ms (units of time are assumed to be milliseconds for this example), yielding a rate of 20Hz. The graphical editor supports the use of *compound models*. A compound model contains an RTDF model, allowing the use of hierarchy in the graphical model. Use of hierarchy aids in keeping the graphical representation manageable. The RTDF model is flattened before it is output to the verification tool. *PreprocessImageN* and *ProcessingPipeN* are compound nodes, where N is a number corresponding to the target classifier index of the computations (N=0,1,2,3).

Figure 29 shows the two-node model contained in *PreprocessImage0*, and Figure 30 shows the seven-node model contained in *ProcessingPipe0*. All *PreprocessImageN* and *ProcessingPipeN* compound models contain the same RTDF model structure. Computation execution time intervals are listed in

Table 1. Processor assignments can be seen in the lower right corner of the computation. All local buffer sizes between computations have a length of one.

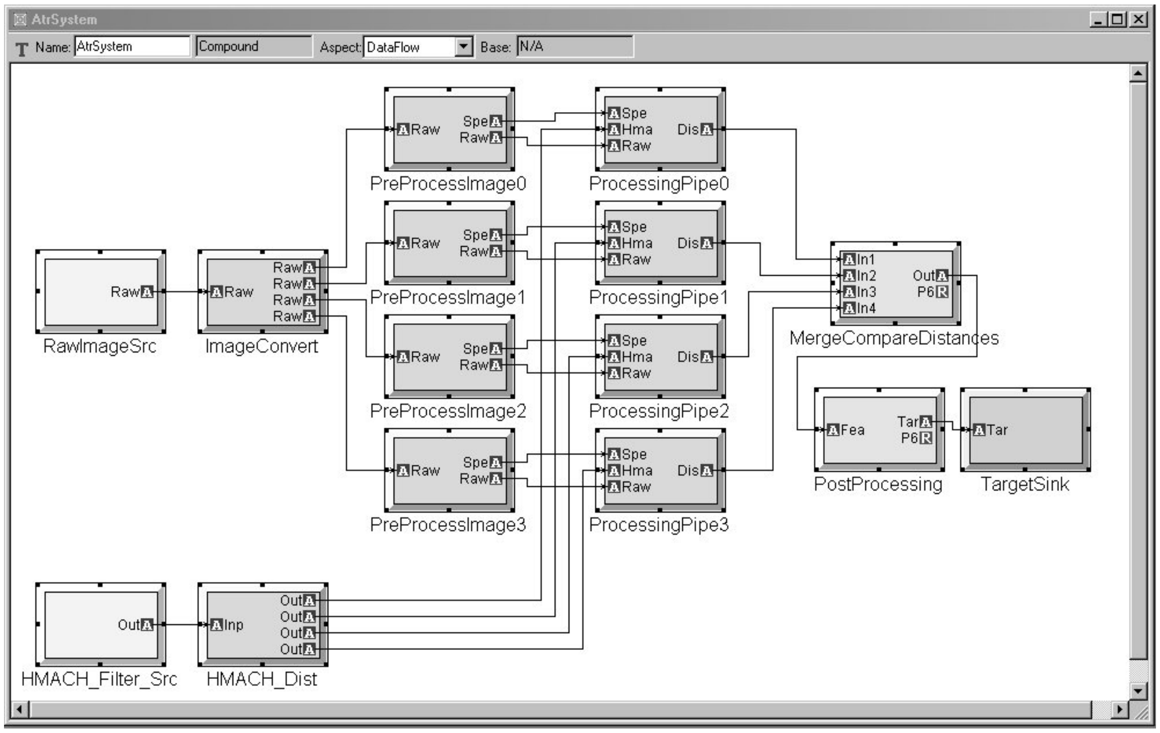


Figure 28: ATR RTDF model

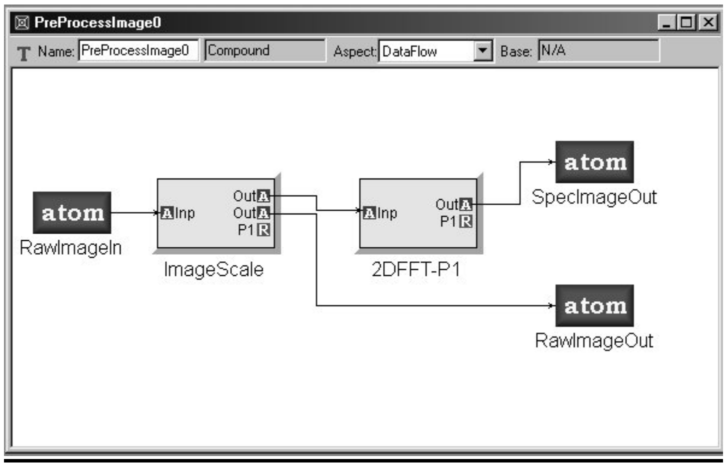
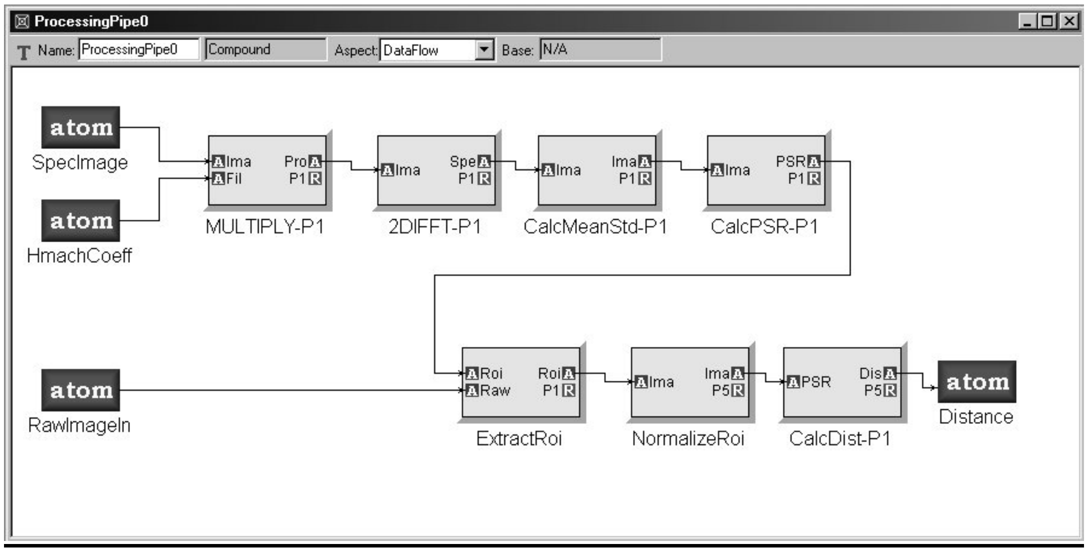


Figure 29: *PreProcessImage* Model


ProcessingPipe0

Name: ProcessingPipe0 Compound Aspect: DataFlow Base: N/A

Figure 30: *ProcessingPipe* Model

Table 1: ATR Computation execution times

Computation	Lower bound execution time (ms)	Upper bound execution time (ms)
ImageScale	4	4
2D-FFT	15	15
MULTIPLY	6	6
2D-IFFT	15	15
CalcMeanStd	3	3
CalcPSR	5	5
ExtractRoi	2	2
NormalizeRoi	3	3
CalcDist	1	4
MergeCompareDistances	8	16
PostProcessing	13	13

The hardware platform for this system is a network of seven processors. Resource Model for this system is shown in Figure 31. All hardware communication links have an internal buffer queue size of two.

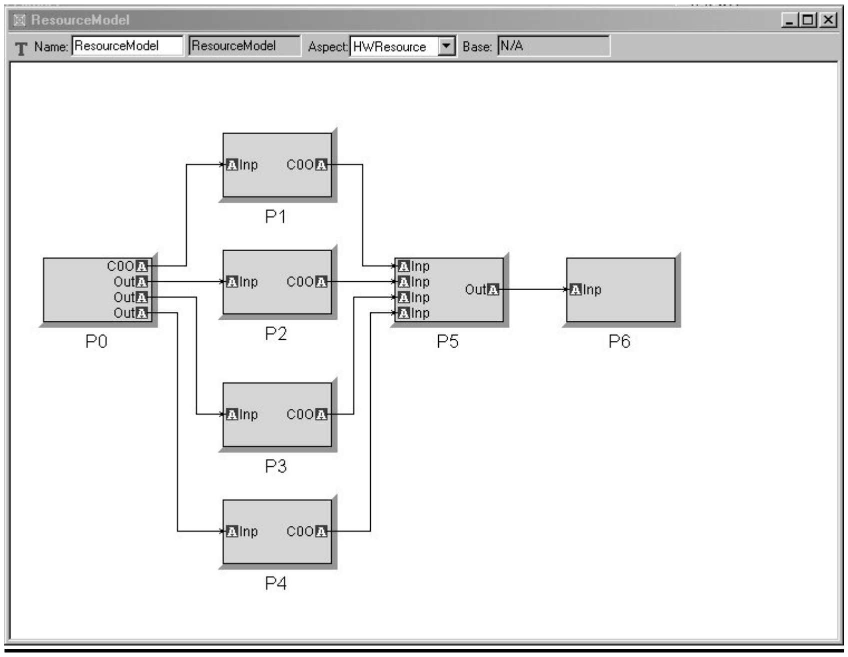


Figure 31: ATR Resource Model

System Verification

Schedulability analysis was performed on this system model and the system was found to not be schedulable. The verification reports the following:

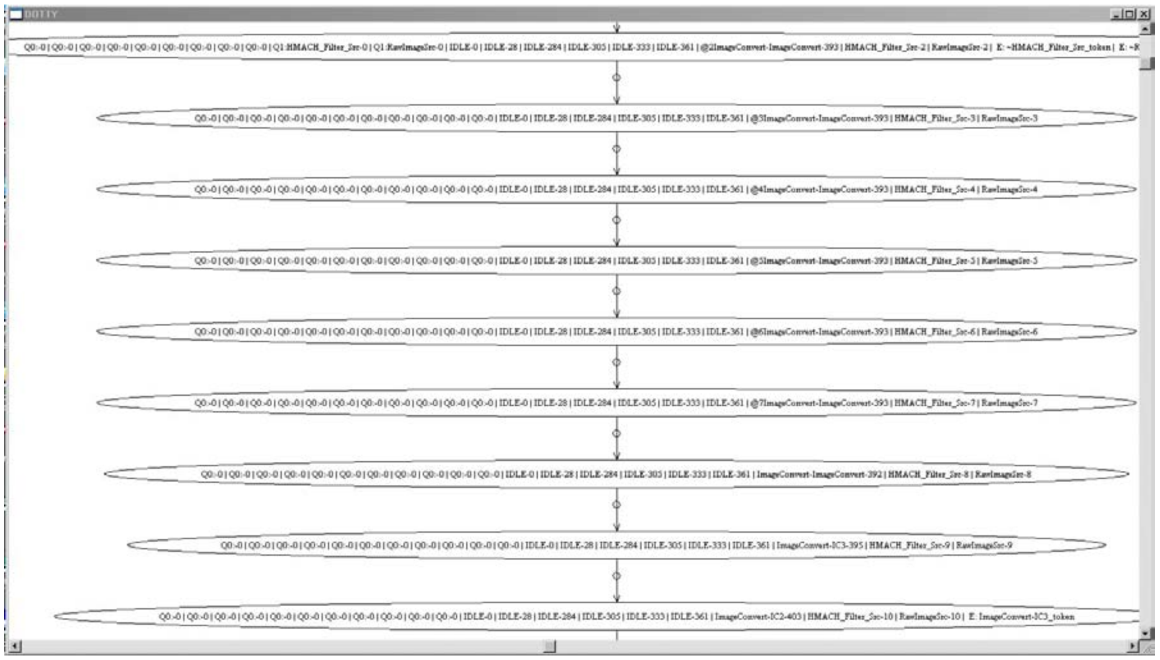
The following exception states can be encountered:

State "OF3:ProcessingPipe2-CalcDist-P3ProcessingPipe1-CalcDist-P2ProcessingPipe0-CalcDist-P1" associated with Communication link "P5_to_P6"

We see that the exception state was a state that indicates that a communication link entered an overflow condition. From the state label we know what the contents of the communication queue were when the overflow occurred. The first item is a token from computation "P2ProcessingPipe0-CalcDist-P1", the second a token from "ProcessingPipe1-CalcDist-P2", and the token that had nowhere to go, and therefore cause the behavior model to signal an exception was from "ProcessingPipe2-CalcDist-

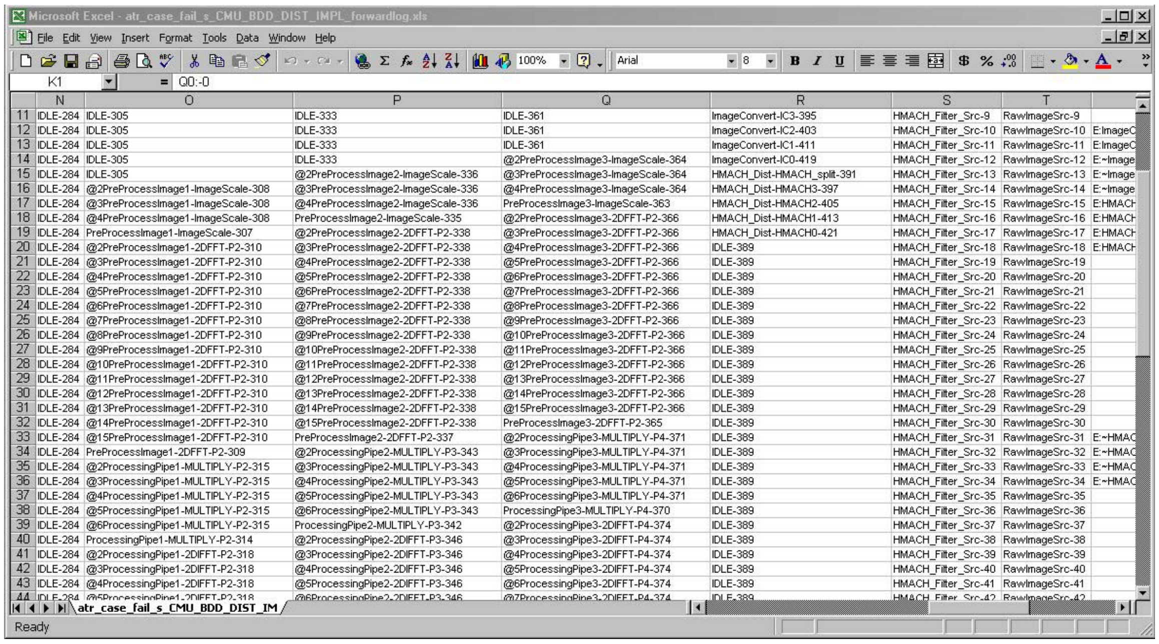
P3”. Also noted is that this exception is associated with communication link “P5_to_P6” (known because the exception state is part of this communication link’s behavior graph).

A counterexample is requested. The counterexample algorithm returns a set of all paths that lead to the first exception state set. This set of paths is returned as a tree structure. This tree structure can be provided to the user in a graphical format as shown in Figure 32 or in a tabular format as shown in Figure 33. The tabular format can be imported into a spreadsheet, such as Microsoft Excel, for a formatted presentation (the tabular is the most readable, but is awkward for counterexample with more than a few paths).



```
Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q1:HMACH_Filter_Src-0 | Q1:RawImageSrc-0 | IDLE-0 | IDLE-28 | IDLE-284 | IDLE-305 | IDLE-333 | IDLE-361 | @2ImageConvert-ImageConvert-393 | HMACH_Filter_Src-2 | RawImageSrc-2 |  E: ~HMACH_Filter_Src_token |  E: ~R
Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | IDLE-0 | IDLE-28 | IDLE-284 | IDLE-305 | IDLE-333 | IDLE-361 | @3ImageConvert-ImageConvert-393 | HMACH_Filter_Src-3 | RawImageSrc-3
Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | IDLE-0 | IDLE-28 | IDLE-284 | IDLE-305 | IDLE-333 | IDLE-361 | @4ImageConvert-ImageConvert-393 | HMACH_Filter_Src-4 | RawImageSrc-4
Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | IDLE-0 | IDLE-28 | IDLE-284 | IDLE-305 | IDLE-333 | IDLE-361 | @5ImageConvert-ImageConvert-393 | HMACH_Filter_Src-5 | RawImageSrc-5
Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | IDLE-0 | IDLE-28 | IDLE-284 | IDLE-305 | IDLE-333 | IDLE-361 | @6ImageConvert-ImageConvert-393 | HMACH_Filter_Src-6 | RawImageSrc-6
Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | IDLE-0 | IDLE-28 | IDLE-284 | IDLE-305 | IDLE-333 | IDLE-361 | @7ImageConvert-ImageConvert-393 | HMACH_Filter_Src-7 | RawImageSrc-7
Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | IDLE-0 | IDLE-28 | IDLE-284 | IDLE-305 | IDLE-333 | IDLE-361 | ImageConvert-ImageConvert-392 | HMACH_Filter_Src-8 | RawImageSrc-8
Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | IDLE-0 | IDLE-28 | IDLE-284 | IDLE-305 | IDLE-333 | IDLE-361 | ImageConvert-IC3-395 | HMACH_Filter_Src-9 | RawImageSrc-9
Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | Q0:-0 | IDLE-0 | IDLE-28 | IDLE-284 | IDLE-305 | IDLE-333 | IDLE-361 | ImageConvert-IC2-403 | HMACH_Filter_Src-10 | RawImageSrc-10 |  E: ImageConvert-IC3_token
```

Figure 32: Counterexample graphical output


Microsoft Excel - atr_case_fail_s_CMU_BDD_DIST_IMPL_forwardlog.xls
File Edit View Insert Format Tools Data Window Help
100%
Arial
8
K1
= Q0:-0
atr_case_fail_s_CMU_BDD_DIST_IM
Ready

	N	O	P	Q	R	S	T	
11	IDLE-284	IDLE-305	IDLE-333	IDLE-361	ImageConvert-IC3-395	HMACH_Filter_Src-9	RawImageSrc-9	
12	IDLE-284	IDLE-305	IDLE-333	IDLE-361	ImageConvert-IC2-403	HMACH_Filter_Src-10	RawImageSrc-10	E:ImageC
13	IDLE-284	IDLE-305	IDLE-333	IDLE-361	ImageConvert-IC1-411	HMACH_Filter_Src-11	RawImageSrc-11	E:ImageC
14	IDLE-284	IDLE-305	IDLE-333	@2PreProcessImage3-ImageScale-364	ImageConvert-IC0-419	HMACH_Filter_Src-12	RawImageSrc-12	E:~Image
15	IDLE-284	IDLE-305	@2PreProcessImage2-ImageScale-336	@3PreProcessImage3-ImageScale-364	HMACH_Dist-HMACH_split-391	HMACH_Filter_Src-13	RawImageSrc-13	E:~Image
16	IDLE-284	@2PreProcessImage1-ImageScale-308	@3PreProcessImage2-ImageScale-336	@4PreProcessImage3-ImageScale-364	HMACH_Dist-HMACH3-397	HMACH_Filter_Src-14	RawImageSrc-14	E:~Image
17	IDLE-284	@3PreProcessImage1-ImageScale-308	@4PreProcessImage2-ImageScale-336	PreProcessImage3-ImageScale-363	HMACH_Dist-HMACH2-405	HMACH_Filter_Src-15	RawImageSrc-15	E:HMACH
18	IDLE-284	@4PreProcessImage1-ImageScale-308	PreProcessImage2-ImageScale-335	@2PreProcessImage3-2DFFT-P2-366	HMACH_Dist-HMACH1-413	HMACH_Filter_Src-16	RawImageSrc-16	E:HMACH
19	IDLE-284	PreProcessImage1-ImageScale-307	@2PreProcessImage2-2DFFT-P2-338	@3PreProcessImage3-2DFFT-P2-366	HMACH_Dist-HMACH0-421	HMACH_Filter_Src-17	RawImageSrc-17	E:HMACH
20	IDLE-284	@2PreProcessImage1-2DFFT-P2-310	@3PreProcessImage2-2DFFT-P2-338	@4PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-18	RawImageSrc-18	E:HMACH
21	IDLE-284	@3PreProcessImage1-2DFFT-P2-310	@4PreProcessImage2-2DFFT-P2-338	@5PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-19	RawImageSrc-19	
22	IDLE-284	@4PreProcessImage1-2DFFT-P2-310	@5PreProcessImage2-2DFFT-P2-338	@6PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-20	RawImageSrc-20	
23	IDLE-284	@5PreProcessImage1-2DFFT-P2-310	@6PreProcessImage2-2DFFT-P2-338	@7PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-21	RawImageSrc-21	
24	IDLE-284	@6PreProcessImage1-2DFFT-P2-310	@7PreProcessImage2-2DFFT-P2-338	@8PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-22	RawImageSrc-22	
25	IDLE-284	@7PreProcessImage1-2DFFT-P2-310	@8PreProcessImage2-2DFFT-P2-338	@9PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-23	RawImageSrc-23	
26	IDLE-284	@8PreProcessImage1-2DFFT-P2-310	@9PreProcessImage2-2DFFT-P2-338	@10PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-24	RawImageSrc-24	
27	IDLE-284	@9PreProcessImage1-2DFFT-P2-310	@10PreProcessImage2-2DFFT-P2-338	@11PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-25	RawImageSrc-25	
28	IDLE-284	@10PreProcessImage1-2DFFT-P2-310	@11PreProcessImage2-2DFFT-P2-338	@12PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-26	RawImageSrc-26	
29	IDLE-284	@11PreProcessImage1-2DFFT-P2-310	@12PreProcessImage2-2DFFT-P2-338	@13PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-27	RawImageSrc-27	
30	IDLE-284	@12PreProcessImage1-2DFFT-P2-310	@13PreProcessImage2-2DFFT-P2-338	@14PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-28	RawImageSrc-28	
31	IDLE-284	@13PreProcessImage1-2DFFT-P2-310	@14PreProcessImage2-2DFFT-P2-338	@15PreProcessImage3-2DFFT-P2-366	IDLE-389	HMACH_Filter_Src-29	RawImageSrc-29	
32	IDLE-284	@14PreProcessImage1-2DFFT-P2-310	@15PreProcessImage2-2DFFT-P2-338	PreProcessImage3-2DFFT-P2-365	IDLE-389	HMACH_Filter_Src-30	RawImageSrc-30	
33	IDLE-284	@15PreProcessImage1-2DFFT-P2-310	PreProcessImage2-2DFFT-P2-337	@2ProcessingPipe3-MULTIPLY-P4-371	IDLE-389	HMACH_Filter_Src-31	RawImageSrc-31	E:~HMAC
34	IDLE-284	PreProcessImage1-2DFFT-P2-309	@2ProcessingPipe2-MULTIPLY-P3-343	@3ProcessingPipe3-MULTIPLY-P4-371	IDLE-389	HMACH_Filter_Src-32	RawImageSrc-32	E:~HMAC
35	IDLE-284	@2ProcessingPipe1-MULTIPLY-P2-315	@3ProcessingPipe2-MULTIPLY-P3-343	@4ProcessingPipe3-MULTIPLY-P4-371	IDLE-389	HMACH_Filter_Src-33	RawImageSrc-33	E:~HMAC
36	IDLE-284	@3ProcessingPipe1-MULTIPLY-P2-315	@4ProcessingPipe2-MULTIPLY-P3-343	@5ProcessingPipe3-MULTIPLY-P4-371	IDLE-389	HMACH_Filter_Src-34	RawImageSrc-34	E:~HMAC
37	IDLE-284	@4ProcessingPipe1-MULTIPLY-P2-315	@5ProcessingPipe2-MULTIPLY-P3-343	@6ProcessingPipe3-MULTIPLY-P4-371	IDLE-389	HMACH_Filter_Src-35	RawImageSrc-35	
38	IDLE-284	@5ProcessingPipe1-MULTIPLY-P2-315	@6ProcessingPipe2-MULTIPLY-P3-343	ProcessingPipe3-MULTIPLY-P4-370	IDLE-389	HMACH_Filter_Src-36	RawImageSrc-36	
39	IDLE-284	@6ProcessingPipe1-MULTIPLY-P2-315	ProcessingPipe2-MULTIPLY-P3-342	@2ProcessingPipe3-2DIFFT-P4-374	IDLE-389	HMACH_Filter_Src-37	RawImageSrc-37	
40	IDLE-284	ProcessingPipe1-MULTIPLY-P2-314	@2ProcessingPipe2-2DIFFT-P3-346	@3ProcessingPipe3-2DIFFT-P4-374	IDLE-389	HMACH_Filter_Src-38	RawImageSrc-38	
41	IDLE-284	@2ProcessingPipe1-2DIFFT-P2-318	@3ProcessingPipe2-2DIFFT-P3-346	@4ProcessingPipe3-2DIFFT-P4-374	IDLE-389	HMACH_Filter_Src-39	RawImageSrc-39	
42	IDLE-284	@3ProcessingPipe1-2DIFFT-P2-318	@4ProcessingPipe2-2DIFFT-P3-346	@5ProcessingPipe3-2DIFFT-P4-374	IDLE-389	HMACH_Filter_Src-40	RawImageSrc-40	
43	IDLE-284	@4ProcessingPipe1-2DIFFT-P2-318	@5ProcessingPipe2-2DIFFT-P3-346	@6ProcessingPipe3-2DIFFT-P4-374	IDLE-389	HMACH_Filter_Src-41	RawImageSrc-41	
44	IDLE-284	@5ProcessingPipe1-2DIFFT-P2-318	@6ProcessingPipe2-2DIFFT-P3-346	@7ProcessingPipe3-2DIFFT-P4-374	IDLE-389	HMACH_Filter_Src-42	RawImageSrc-42	

Figure 33: Counterexample tabular output

We know the communication link between processor P5 and P6 became full and was unable to accept a token from computation *ProcessingPipe2-CalcDist-P3*. An examination of the counterexample reveals the problem. There is a case where the *CalcDist* function from each of the four processing pipes may output a data token at nearly the same time, resulting in an overload of the communication link from processor P5 to processor P6.

Looking at the model, we see that eight computations are allocated to processor P5: four *NormalizeRoi* computations and four *CalcDist* computations, each from their respective *ProcessingPipe* compound models. The priorities of these computations are such that all *NormalizeRoi* computations have priority over all *CalcDist* computations. The counterexample shows that all *NormalizeRoi* computations receive tokens at nearly the same time. Because all *NormalizeRoi* components have priority over other computations on the processors, they will all execute in series. Next, all *CalcDist* computations will execute in series. As we can see from

Table 1, each *CalcDist* computation has an execution time interval of [1,4]. The counterexample shows that the single case where all *CalcDist* computations have an execution time of 1ms causes an overload of the communication link. To prevent this, we can adjust the computation's priorities such that each *CalcDist* computation has a higher priority than his upstream *NormalizeRoi* computation.

This change was made to the system model and the schedulability analysis was run again. The tool reports the system is schedulable in all cases.

Verification Performance

The size of the distributed next-state relation is shown in Table 2 for both an OBDD and a MTBDD implementation. The graph sizes are given for each component of the distributed next-state relation and the total number of nodes for all components is shown. The total number of MTBDD nodes is 43712 vs. 38948 OBDD nodes. It is interesting to note that the graph sizes are of nearly the same size, although the representation is significantly different.

The number of OBDD/MTBDD nodes at each step in the reachability calculation is shown in Figure 34 and Figure 35. The iteration in Figure 34 stops at 81 steps when an exception state is reached. The iteration in Figure 35 continues until a fixpoint is reached. Although the MTBDD total system representation was slightly larger than the OBDD representation, the MTBDD representation provides better performance in the size of the state set at each step in the reachability set calculation. In general, the size of the representation of the reachability set will be the bottleneck of system analysis. It is interesting to note the changes in the rate of growth as non-deterministic model features are encountered in the traversal.

Table 2: TRSchart and corresponding symbolic representation sizes

	TRSchart states (BDD)	BDD nodes	TRSchart states (MTBDD)	MTBDD Nodes
Communication Links:				
P4_to_P5	4	15	4	42
P0_to_P4	15	94	14	256
P5_to_P6	85	1429	85	3016
P3_to_P5	4	15	4	42
P0_to_P3	15	94	15	256
P0_to_P2	15	94	15	256
P0_to_P1	15	94	15	256
P2_to_P5	4	15	4	42
P1_to_P5	4	15	4	42
C_HMACH_Filter_Src_HMACH_split	3	12	3	33
C_RawImageSrc_ImageConvert	3	12	3	33
Processors:				
P5	846	25571	256	29349
P4	184	1878	28	1473
P6	91	1204	21	941
P0	48	2298	36	2719
P3	184	1894	28	1658
P2	184	1889	28	1637
P1	184	1869	28	1643
P_HMACH_Filter_Src	51	228	2	9
P_RawImageSrc	51	228	2	9
	Total Nodes:	38948	Total Nodes:	43712

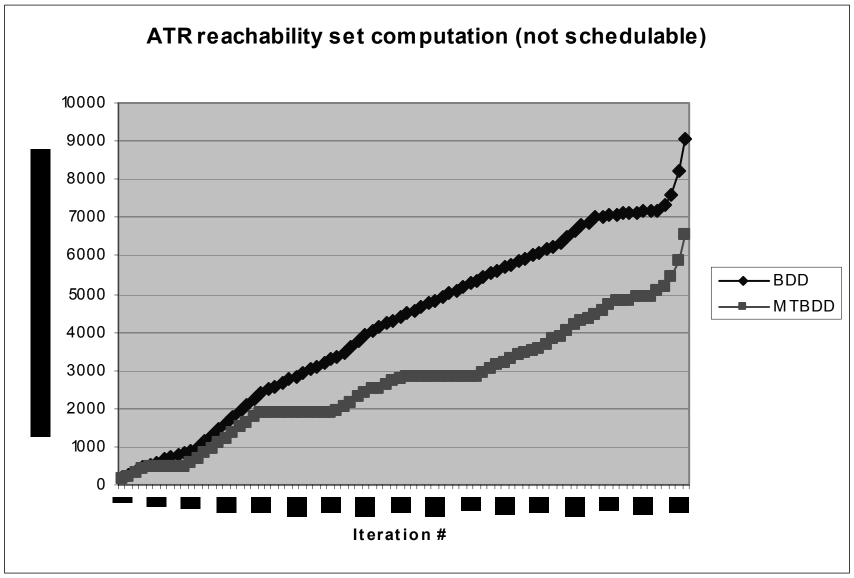


Figure 34: OBDD/MTBDD sizes for reachability set computation

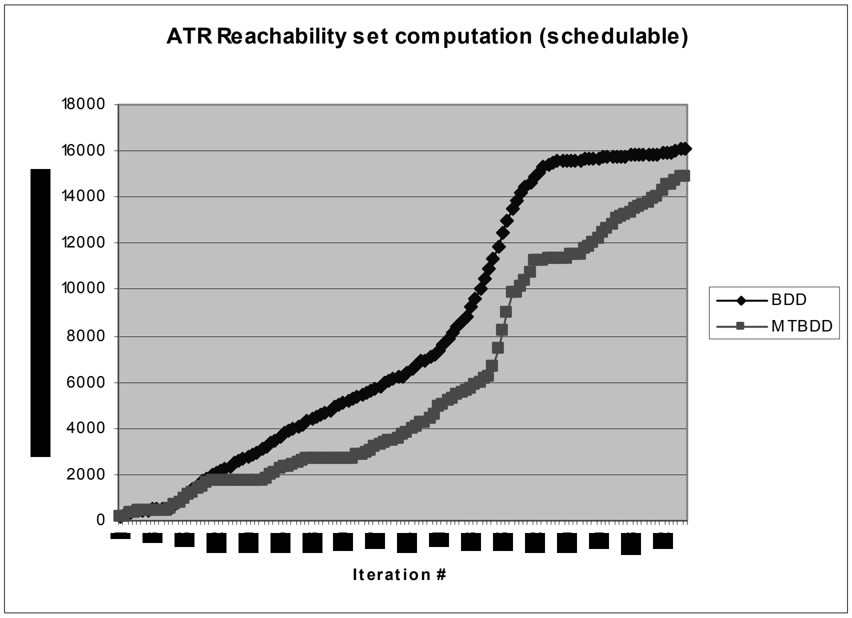


Figure 35: OBDD/MTBDD sizes for reachability set computation (corrected model)

It is not always the case that MTBDDs will outperform OBDDs. For the models analyzed with this tool, MTBDDs seem to perform best on systems where states have a fixed time duration, rather than an interval. As the number of states with time intervals in concurrent sub-TRScharts increase, the performance of the MTBDDs decreases. This seems to be due to the augmentation of the MTBDD representation shown in Chapter IV where MTBDDs were modified such that a terminal node could contain a set of Boolean vectors. The need to represent more than one evaluation increases as the need to represent a variety of timed states at many time procession values in concurrent states.

Integrated Vehicle Health Management System

To evaluate the scalability of the verification tool, an example system has been modeled. Subsets of this system are also modeled to provide data on the growth rate of the symbolic representation with respect to application growth. The system is a health monitoring system used for analysis of Space Shuttle Main Engine (SSME) data. This system provides on-line monitoring of SSME turbopump accelerometer data for early detection of a catastrophic pump failure event. This application is implemented on a multi-processor platform.

IVHM Model

The system must process 32 channels plus one key phasor channel of continuous, sampled accelerometer data sampled at a rate of 6.6 ksamples/sec per channel (corresponding to a token generation period of 150ms). This data is sent to the health monitoring system as a single data stream. Each channel of data is distributed to its respective analysis algorithms in blocks of 1024 samples by the *DataMgr* and *DataDist* computations. These computations are shown in the RTDF model of the overall system in Figure 36 with a zoomed-in view shown in Figure 37. The *DataMgr* and *DataDist* computations also separate out another channel of data, labeled “key phasor” from the input data block and copies and distributes this data along with the data blocks to all *ChanN_processing* compound models, where N is the channel number.

Each *ChanN_processing* compound contains a set of computations for a single channel (see Figure 38). The processing begins with a *EU_Conv* computation performing an engineering unit conversion on a data block. This transforms the data from a set of raw digitized sample values into a meaningful unit of measure. A 1024-point FFT is performed on the data, and then the complex spectral data is converted into a squared magnitude format with the *1K_FFT* and *Mag_Sqr* computations, respectively. The spectral squared magnitude data is passed to the *FindPeaks* computation that extracts at most the top eight peaks above a given threshold. The *FindPeaks* execution time is data dependent, depending on the number of peaks present in the spectral data block. The *CPLE* (Coherent Phase Line Enhancer) computation takes engineering unit converted time data and *KeyPhasor* data as input. The *KeyPhasor* is a speed signal indicated as a sine wave with the same frequency as (and in phase with) the turbopump rotation. Briefly, the CPLE produces a single coherence value that gives an indication of how strongly related the phasing of the current synchronous peak is to the turbopump shaft. Spectral data from the *1K_FFT* is also sent to the *SyncTracking* computation where the amplitude of the maximum peak in a window around the current rotational speed (frequency) of the pump shaft is tracked.

Tracking data from *SyncTracking* computation, the coherence value from the CPLE computation, and EU converted time data is sent to *SensorVal*. *SensorVal* uses this information to determine the health of a sensor. Returning back to the top level of the model hierarchy, we see that the results from the tracking algorithm and sensor validation algorithm from all channels in a group of four is sent to the *RedlineLogic* computation. This computation can produce a signal for an engine cutoff if a specified number of peak tracking values are out of range, and the appropriate sensors are found to be valid. Finally, we have a *Storage* computation. This computation stores spectral peaks and engineering unit converted time data to non-volatile storage.

All computations are of type IF_ALL. All local on-processor buffers between computations are of size one. All communication link queues have a buffer of length two. Computation execution times are listed in Table 3.

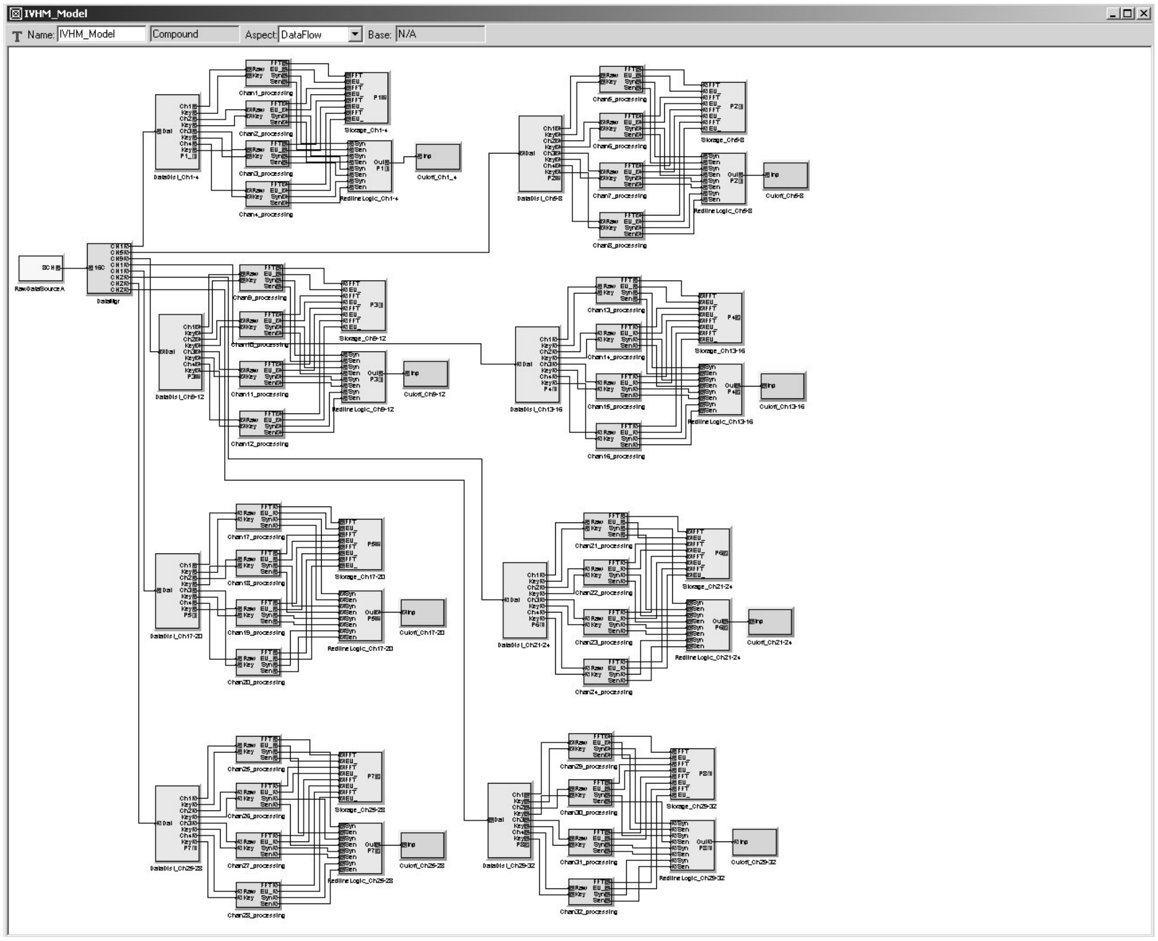


Figure 36: IVHM RTDF Model

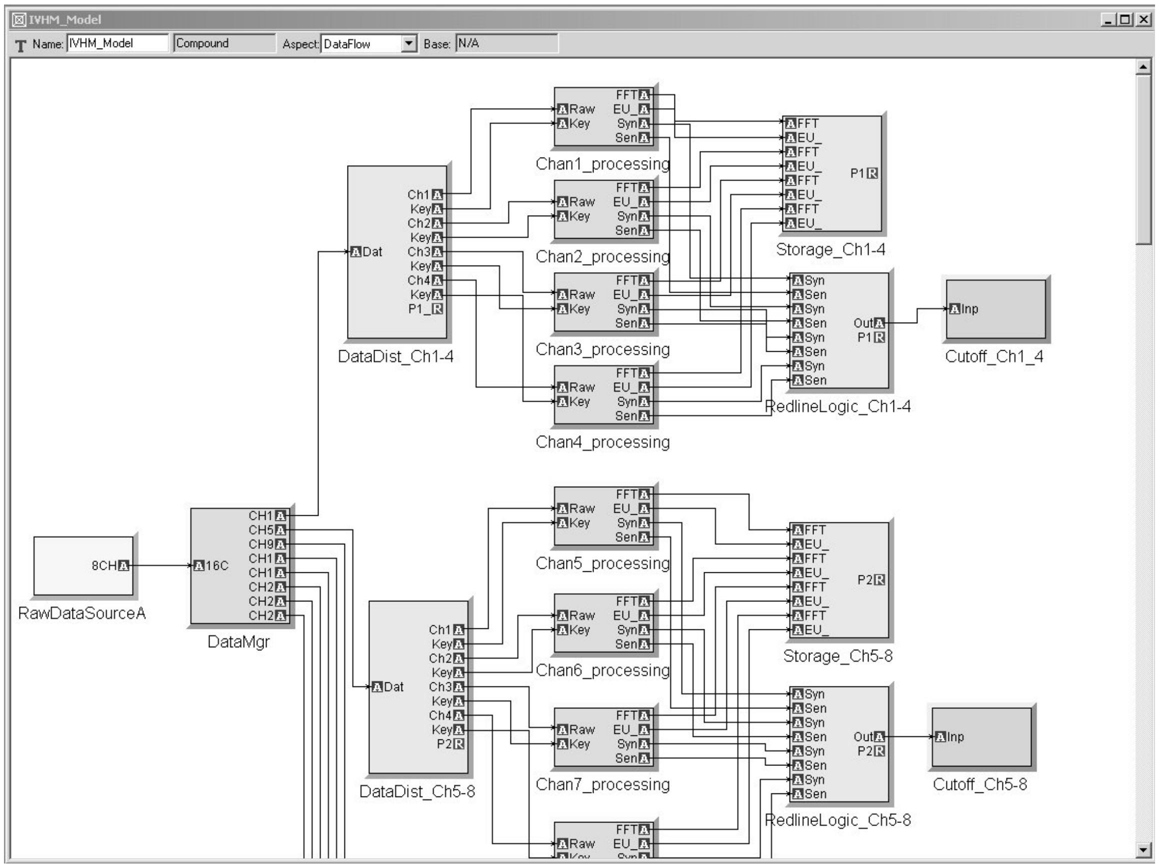


Figure 37: A closer view of channels 1-4 processing

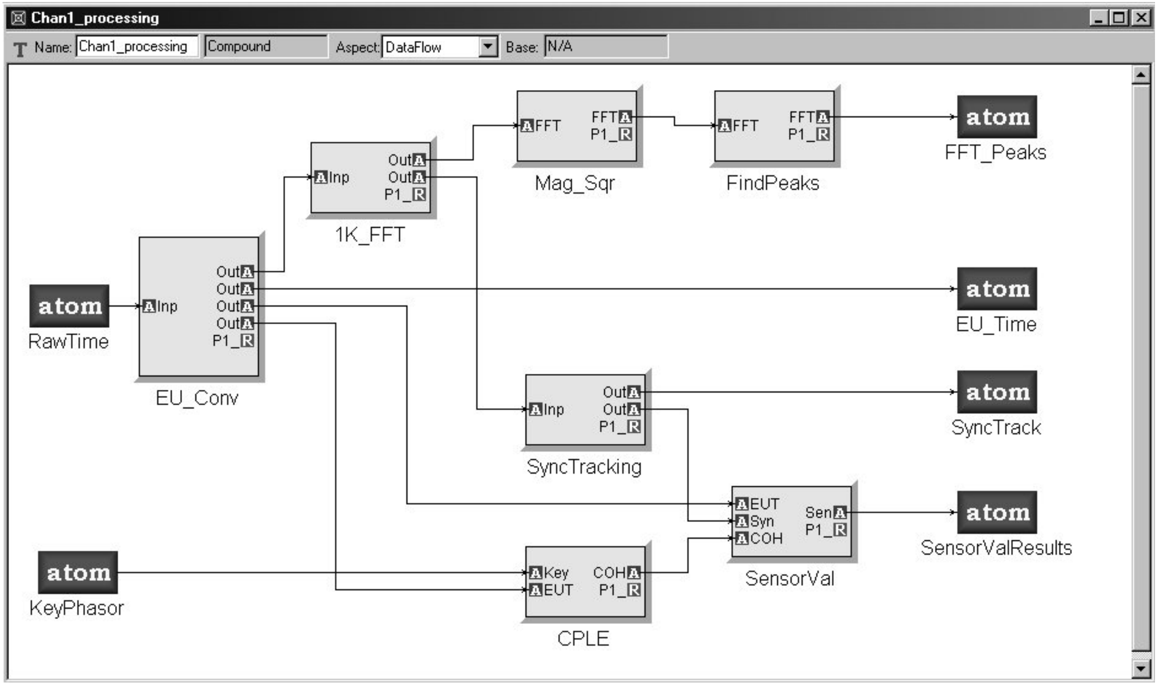


Figure 38: Chan1_processing compound model - processing of a single channel

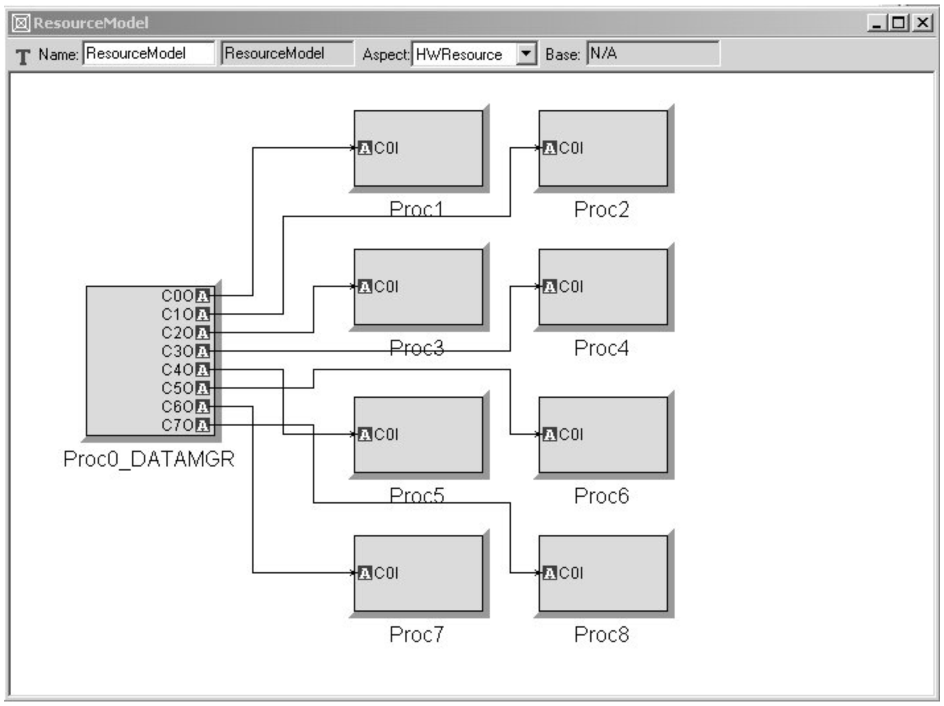


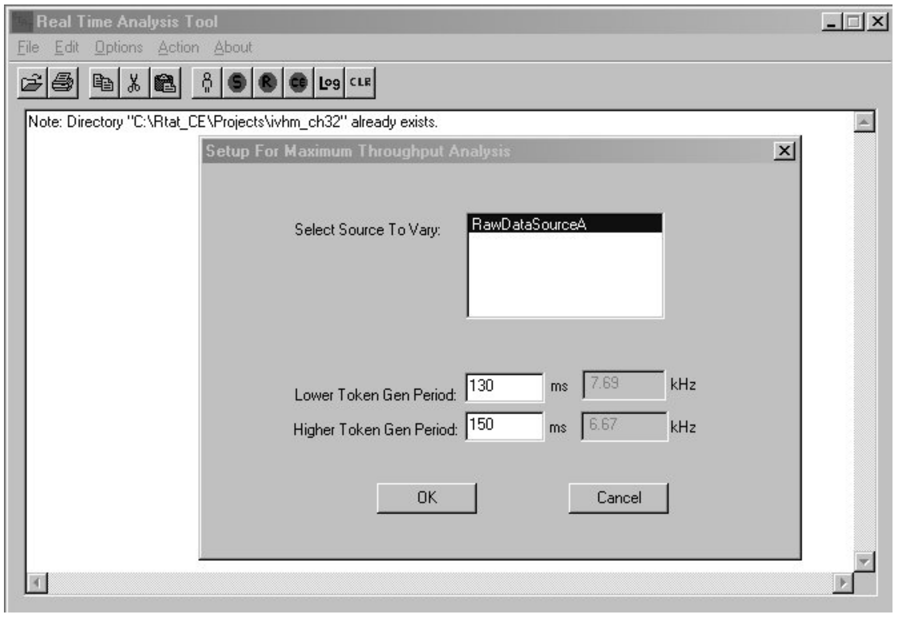
Figure 39: IVHM Resource Model

Table 3: IVHM computation execution times

Computation	Lower bound execution time (ms)	Upper bound execution time (ms)
DataMgr	4	4
Data-Dist	4	4
EU-Conv	6	6
1K_FFT	7	7
Mag_Sqr	4	4
FindPeaks	2	6
SyncTracking	3	3
SensorVal	1	5
CPLE	2	7
Storage	1	1
RedlineLogic	1	1

System Verification

The schedulability analysis was performed on the IVHM model. The system was found to be schedulable for a data input period of *RawDataSourceA* of 150ms (rate of 6.6 ksamples/sec). An important performance question is: "what is the highest input rate at which the system meets schedulability requirements?" To answer this, throughput analysis was performed. You must select the source node of the model you wish to vary and the upper and lower bounds of the token generation period you want to explore. A screen capture of the throughput setup dialog of the verification tool is shown below in Figure 40. The throughput analysis reports that a minimum period of the source node *RawDataSourceA* resulting in a schedulable system is 147ms (a rate of 6.8 ksamples/sec).



Note: Directory "C:\Rtat_CE\Projects\ivhm_ch32" already exists.

Setup For Maximum Throughput Analysis

Select Source To Vary: RawDataSourceA

Lower Token Gen Period: 130 ms 7.69 kHz

Higher Token Gen Period: 150 ms 6.67 kHz

Figure 40: Verification tool throughput analysis dialog

Verification Tool Performance

The model for the IVHM system was built up incrementally in groups of four channels worth of computations at a time to show the growth rate of the symbolic representation as the system size increases. The results show the schedulability analysis algorithm scales well as the model is expanded to handle more processors as shown in Figure 41 and Figure 42. The size of the symbolic representation of the transition relation is shown below as a function of increasing number of processors. Each processing node is executing a similar RTDF model. As seen from Figure 41 and Figure 42 the size of the transition relation scales linearly as the system size increases with additional processors. This linear scaling in the symbolic OBDD/MTBDD representation is due to the use of a "distributed" representation of TRSchart/RSchart models (discussed in Chapter IV). This distributed relation maintains the representations of the sub-components as separate entities. Total memory usage of the verification tool for any of the system model variations did not exceed 70Mb.

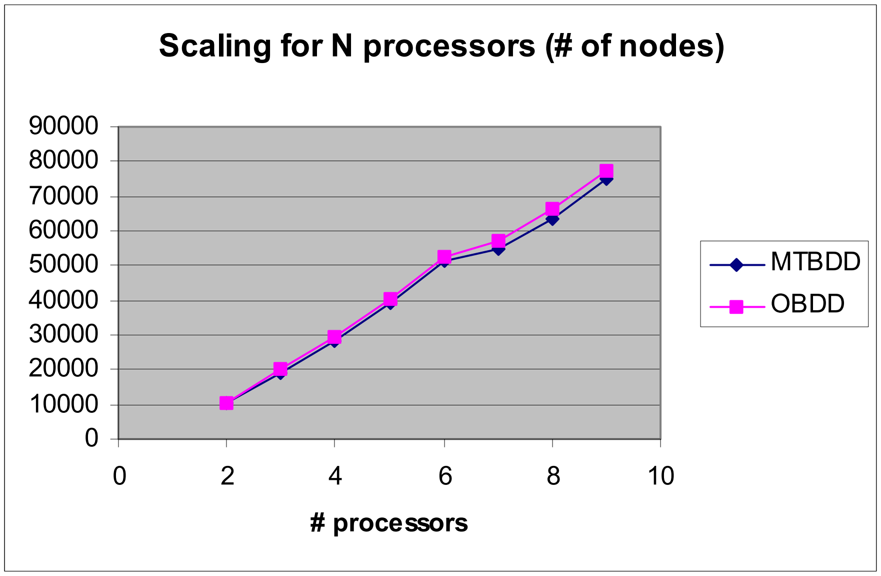


Figure 41: Transition relation scaling: number of nodes vs. number of processors

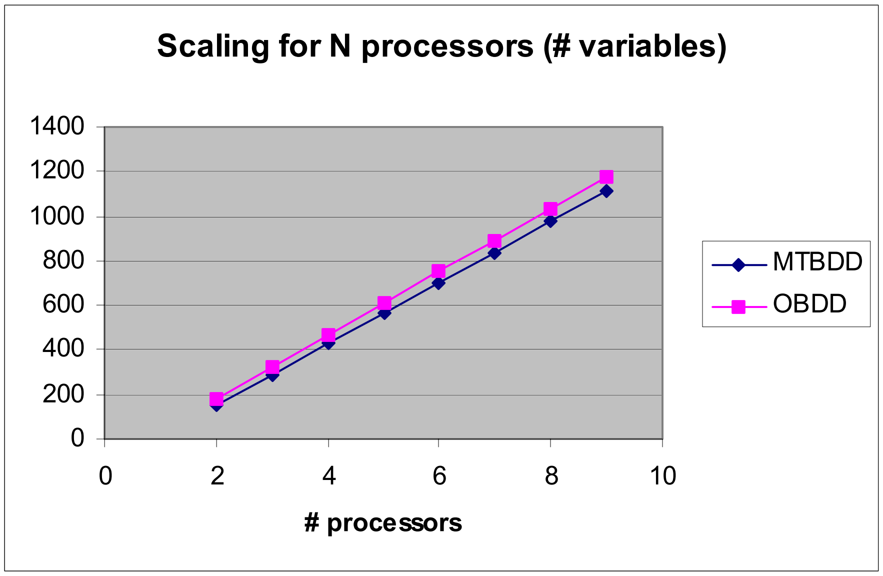


Figure 42: Transition relation scaling: number of variables vs. number of processors

Comparison to other tools

It is possible to perform this analysis with other verification tools that are publicly available. Two such tools are Kronos and UPPAAL described in Chapter II. Both tools use Timed Automata as their input model description. Creating a timed automata model by hand to represent any sizable system would be a difficult and error-prone task. An automatic behavior model generation scheme similar to the one shown in this dissertation would need to be necessary.

This verification tool performs verifications specific to multi-processor systems with computations modeled with a dataflow model. Use of another tool would require specification of system properties to be verified to be in a temporal logic (TCTL for Kronos or subset of CTL providing reachability constructs for UPPAAL). These properties would need to be automatically generated along with the behavior model.

The timed automata representation is capable of modeling complex interactions between models and representation of multiple system clocks (and variables for

UPPAAL) in a continuous time domain. Although providing a wider range of applications that can be modeled, the expressiveness of the model limits the size of the systems these tools can verify. The ability of these tools to successfully verify a model depends on the size of the model and the number of clock variables used [55]. For example, in [61], Havelund presents a case study of verification of an audio/video control protocol responsible for transfer of multiple messages over a single bus that must provide collision detection. The model of the system is a composition of 9 timed automata models, each with fewer than 20 locations each. Complete verification of a reachability specification with UPPAAL required 30 minutes and 90MB of memory on a Sparc 10.

The tool presented in this work translates the system model into a single finite-state model that represents the entire system behavior. Because of its simplicity, the simple finite-state model can be analyzed entirely in its symbolic form. TRSchart models composed of sub-TRSchart models with ~30,000 states each have been successfully represented symbolically and verified with the tool presented in this work.

CHAPTER VII

RESULTS AND FUTURE RESEARCH

Results

The design complexity of embedded, real-time systems is ever increasing. Many of these systems are also *safety-critical systems*, where proper operation could result in loss of property or injury (most safety-critical systems are also real-time systems [56]). Automated tools are needed for design verification of embedded real-time systems. Traditionally, informal methods such as ad hoc testing and simulation methods have been used for system verification, but their results cannot insure *complete* system verification unless *all* system behaviors can be examined.

An approach to verification of signal-processing, real-time, multi-processor systems is presented in this dissertation. Multi-processor systems have complex interactions between components of the system. Also, systems often have non-deterministic characteristics that cannot be resolved at the level of abstraction at which the system is modeled. It is not feasible to verify correct system behavior via simulation. The number of possible behaviors is simply too large to enumerate. Symbolic model checking methods are used enabling efficient verification of complex systems with large state spaces. A verification tool has been developed implementing the verification framework presented in this dissertation.

RScharts, a new statechart-like model with synchronous step semantics, was defined. A method for the symbolic modeling of RScharts using OBDDs allowing non-deterministic behavior has been developed. A distributed technique for representation of

RScharts using OBDDs has been developed to solve problems encountered when representing large state space systems.

An extension to the RSchart model with added timed semantics, called TRScharts, was defined. A novel MTBDD-based symbolic representation of timed, finite-state systems has been developed. State timing information is captured in the leaf nodes of the MTBDD structure in an effort to avoid using costly state-expansion techniques used in OBDD based model. This representation has been extended for use with concurrent, timed, finite-state systems as well. This representation is well suited for the efficient representation TRSchart models.

Algorithms have been implemented to symbolically traverse the both the OBDD and MTBDD representations of the RSchart/TRSchart models. Reachability algorithms have been implemented to exhaustively search the state space to find the reachable state set of the system (given an initial state set).

Work from this research has been used in a project with Sandia National Laboratories. The State Space Analysis Tool (SSAT) was created for the exhaustive verification of system safety, security, and reliability properties [57, 58, 59]. The RSchart model is the input language of this tool that is used to model the behavior of physical systems. A symbolic representation of RScharts using OBDDs (presented in Chapter IV) is used as the analysis and simulation engine of the SSAT Tool.

Future Research

Currently, a TRSchart object structure is being built representing the global behavior of the system though the composition of the behavior spaces of the system components discussed in previous sections. The symbolic OBDD/MTBDD representation is built directly from the RSchart/TRSchart model. It is possible to build the OBDD/MTBDD structure directly "on-the-fly" from the system model while bypassing the step of creating a TRSchart object structure, but preserving adherence to RSchart/TRSchart semantics. This method of construction may be necessary if the state-space of the processor and communication link behavior models is prohibitively large to build explicitly.

Another area that affects the scalability of the verification methods presented in this work is the size of the communication link behavior graphs. The size of a communication link behavior graph can become extremely large as the number of token flows multiplexed across a communication links and/or its depth grows. Some other representation may be needed if large queue depths combined with many data flows multiplexed across the communication link must be modeled.

Currently, the primary focus of the tool is schedulability verification. The range of properties that can be proven correct can be extended by the development of a more general timed specification language such as Clocked CTL (CCTL) [60] so that general safety and real-time properties can be expressed and verified. At the user level, the tool could provide for more application-specific verification, such as a verification of minimum or maximum input to output latency from a specific source node to a specific sink node.

The RTDF dataflow model and verification tool can easily be extended to support multiple token consumption and multiple token generation for all computation inputs and outputs as found in the SDF model of computation.

The system model can be extended to more accurately model the system. Currently, tokens represent some generic piece of data with no notion of the size of this data. Tokens could be assigned a data type. Each data type would have its own set of parameters indicating the size of the data the token represents. All computation inputs and outputs could be assigned valid token types it may accept and generate. This scheme has the advantage of providing type checking for the purpose of enforcing compatibility of token transfers between computations.

Currently, communication links only model the transfer of tokens. All physical communication mediums have a finite rate at which they transfer data. The communication link behavior model should model the effects of this token transfer latency.

Signal Processing systems often perform tasks other than pure dataflow processing. The dataflow model is not able to represent reactive system behavior or complex control-flow processing. Other models of computation could be added to the verification framework to expand the types of systems that may be analyzed.

It would be possible to add a higher-level optimization (design space exploration) utility that could use this tool as an engine to explore variations of a given system implementation to optimize certain system cost/performance metrics. This tool could provide the means necessary to explore performance extremes for each point design under test.

APPENDIX A

SYMBOLIC MANIPULATION ALGORITHMS

Pre-Image Computation (Backwards Step)

A Pre-Image Computation is used to compute the "previous" set of states from a current set of states. This algorithm is very similar to the Image computation as shown in Equation 4. Let X = $\{x_0,\ldots, x_{n-1}\}$ be the set of variables used to represent a state and X' = $\{x'_0,\ldots, x'_{n-1}\}$ be another set of state variables. A transition relation, TR(X, X'), exists consisting of terms identifying mappings from one system state to another.

We begin with a current state set represented by S(X). First, a variable re-naming is performed, so that the OBDD representing the current state set, S(X), is represented in terms of next state variables, S(X'). Next, the conjunction is performed on S(X') and the next-state transition relation TR(X, X'). The result is the set of all transitions that could have led to the current state set S(X). The last step is to existentially quantify out the next-state set of variables $\{x'_0,\ldots, x'_{n-1}\}$. What we are left with is the source state set of all transitions that could lead to our original state set, S(X).

$$\text{Pre-Image}(\mathrm{S}(X)) = \left(\exists_{X'} \left(\mathrm{S}(X)\big|_{X\rightarrow X'} \wedge \mathrm{TR}(X, X')\right)\right)$$

A pre-image computation is used in counterexample generation for calculating the set of all states reachable from a backwards traversal from a given state or state set.

Zero Unused Output Events

When an Image Computation is performed on a distributed RSchart/TRSchart symbolic representation (OBDD/MTBDD), the resulting set of global configurations only contain an event variable to represent when an action has produced an output event. For any global configuration where an event is not part of that configuration, the variable representing that event will be absent. In order to correctly represent that the event is not present in that configuration, we need to show that the event is absent from the global configuration. Leaving an event variable a "don't care" value would indicate that the event may or may not be present in the global configuration and would indeed represent two valid global configurations, one with the event present and one without.

The following algorithm, shown below in Algorithm 12, has been developed to modify a function such that if a satisfying expression does not depend on a variable, x, (i.e. that variable has a "don't care" value for that expression) then the satisfying expression is changed such that for the expression to evaluate to true, x must be false. If the satisfying expression does depend on x, then the expression is not changed.

The input to an Image Computation function requires that the OBDD/MTBDD representing the "current" global configuration set have all events specified as present or not present in each global configuration in the set. In terms of an OBDD/MTBDD representation, this means all event variables must be fully specified. No event variables may have a "don't care" value. If "don't care" is the value of an event in a global configuration, then that global configuration would really represents two global configurations. One configuration with the event e is present and one configuration where the event is not represent.

```
ZeroDontCareVar(bdd GC, bdd variable e)
// GC is bdd representing set of global configurations, e is bdd of event var
{
        CGSet_NeedsFix = $\forall_e$ GC
        CGSet_NeedsNoFix = GC $\wedge \neg$ CGSet_NeedsFix
        CGSet_FixedEventsBdd = CGSet_NeedsFix $\wedge \neg$ e
        return  CGSet_NeedsNoFix $\vee$ CGSet_FixedEventsBdd
}
```

Algorithm 12: ZeroDontCareVar algorithm

APPENDIX B

ANALYSIS ALGORITHMS

Throughput Algorithm

A method for finding the maximum rate an input signal generate input token while maintaining a schedulable system is presented. The algorithm searches in the range *of high_period* to *low_period*. At each iteration, the TRSchart model is modified. The symbolic representation of the TRSchart is created. Next the schedulability verification is performed. This algorithm returns the minimum period of the selected source, c, for which the system is schedulable. If no period within this range produce a schedulable system, then a NULL value is returned.

```
FindMaxThroughput(Source s, integer high_period, integer low_period)
{
        last_successful_period = NULL
        for(i=high_period; i≥ low_period; i--)
        {
                Modify TRSchart model source node s to have period = i
                Build symbolic representation
                bool schedulable = SchedulabilityVerification()
                if(schedulable = false) return last_successful_period
                else if(schedulable = true) last_successful_period = i
        }
        return last_successful_period
}
```

Counterexample Algorithm

Input: Initial global configuration
Input: bdd representing the set of all exception states
Output: A tree structure representing all paths from initial global configuration
to first encountered exception state(s)

```
PathTree CounterExample(bdd $S_{INIT}$, bdd ExceptionStates)
{
    // Phase I: find first exception state-traverse forward until exception is found
    $S_1 = S_{INIT}$
    AllQueueOverflowStateSet = { set of all exception states }
    i=0;
    do
    {
        $S_2$ = ImageComputation($S_1$, GlobalStateChart);
        if( (ExceptionStates $\cap$ $S_2$) $\neq \emptyset$)
        {
            overflow_state_detected = true;
            GlobalConfigSetWithExceptionState = $S_2 \cap$ ExceptionStates;
        }
        $S_1 = S_2$;
        i++;
    } while( exception _state_detected = false);
    if(!exception_state_detected) return NULL;
    max_back_depth = i;

    // Phase II: Find backward reach set from first exception global config set
    bdd BackReach = BackwardsReachability(GlobalConfigSetWithExceptionState,
    max_back_depth);

    // Phase III: Traverse forward, using backward reachability set to prune
    // This traversal is the counterexample path(s) - path information is captured
    let PathNode be an object that contains a global config and pointer to its children
    TopPathNode = { $S_{INIT}$ };
    BFS_GCList = BddToGlobalConfigList($S_{INIT}$, GlobalStateChart);
    i=0;
    do
    {
        iterate over BFS_GCList
        {
            $G_C$ = current list item;
            $S_1$ = GetGlobalConfigBdd($G_C$);
            $S_2$ = ImageComputation($S_1$, GlobalStateChart) $\cap$ BackReach;
            GCList = BddToGlobalConfigList($S_2$, GlobalStateChart);
            Append GCList to existing NextGCList;
            iterate over GCList
            {
```

```
                        create a new pathnode pn = {GC item};
                        path_node_cur->AddChild(pn);
                    }
                    i++;
                }
            }
            while( i ≤ max_iterations and NextGCList.size > 0 );
            return TopPathNode;
}
```

REFERENCES

[1] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 1999.

[2] Ramming, F., "Mixed Modeling and Simulation of VLSI systems," Logic Design and Simulation, Elsevier Science Publishers B.V., 1986.

[3] Turley, J., "Embedded Processors by the Numbers," *Embedded Systems Programming*, May 1999.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. H. Hwang, "Symbolic model checking: 10^{20} states and beyond", *In Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428-439, 1990.

[5] Hachtel, G., Somenzi, F., Logic Synthesis and Verification Algorithms, Kluwer Academic Publishers, 1996.

[6] Kropf, T., Introduction to Formal Hardware Verification, Springer, 1999.

[7] Stern, U., Dill, D., "Automatic Verification of the SCI Cache Coherence Protocol," *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[8] Miller, S., Srivas, M., "Formal Verification of the AAMP5 Microprocessor – A Case Study in the Industrial Use of Formal Methods," *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 5-8, 1995.

[9] McMillan, K., "The SMV system DRAFT," February 2, 1992.

[10] Bryant, R., "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8, August 1986, pp. 677-691.

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

[12] Meinel, C., Slobodová, A., "Speeding up Variable Orderings of OBDDs," Technical Report 96-40, Computer Science Department, University of Trier, 1996.

[13] Bryant, R., "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification," *International Conference on Computer-Aided Design ICCAD '95*, November 1995, pp. 236-243.

[14] Clarke, E., Fujita, M., Zhao, X., "Hybrid Decision Diagrams – Overcoming the Limitations of MTBDDs and BMDs," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-95-159, April 1995.

[15] Clark, E., Fujita, M., Zhao, X., "Applications of Multi-Terminal Binary Decision Diagrams," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-95-160, April 1995.

[16] Meinel, C., Theobald, T., Algorithms and Data Structures in VLSI Design, Springer-Verlag, 1998.

[17] Wegener, Ingo, Branching Programs and Binary Decision Diagrams, Society for Industrial and Applied Mathematics, 2000.

[18] *bddlib* BDD library available at: http://www-2.cs.cmu.edu/~modelcheck/bdd.html

[19] Somenzi, F., CUDD: CU Decision Diagram package – release 2.1.2, Department of Electrical and Computer Engineering – University of Colorado at Boulder, April 1997. ftp://vlsi.colorado.edu/pub/

[20] Sentovich, E., "A Brief Study of BDD Package Performance," *First International Conference, FMCAD '96*, Formal Methods in Computer-Aided Design, pages 389-403, Springer-Verlag, 1996.

[21] Jahanian, F., Stuart, D., "A Method for Verifying Properties of Modechart Specifications," *Proceedings of the 9th Real-Time Systems Symposium*, IEEE, New York, 1988.

[22] Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri M., "NuSMV : A New Symbolic Model Verifier," In Lecture Notes in Computer Science (CAV 1999), number 1633, pages 495-499, Trento, Italy, Springer, July 1999.

[23] J.R.Burch, E.M. Clarke, D.E. Long, K.L. McMillan, D.L. Dill, "Symbolic Model Checking for Sequential Circuit Verification", Technical Report CMS-CS-93-211 October 1991.

[24] Cimatti, A., Clarke, E., et al., "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," *In Proceedings of International Conference on Computer-Aided Verification (CAV 2002)*, Copenhagen, Denmark, July 27-31, 2002.

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

[26] Campos, S., "Verus 0.9 - Reference Manual," March 1997.

[27] Campos, S., Clarke, E., Marrero, W., Minea, M., “Verus: A Tool for Quantitative Analysis of Finite-State Real-Time Systems,” *ACM Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, CA, June 1995.

[28] Bengtsson, J., et al., "UPPAAL -- a Tool Suite for Automatic Verification of Real-Time Systems," In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, New Brunswick, New Jersey, October 22-24, 1995.

[29] Larsen, K., Pettersson, P., Yi, W., UPPAAL in a Nutshell. *In Springer International Journal of Software Tools for Technology Transfer 1(1+2)*, 1997.

[30] Larsen, G., Pettersson, P., Yi, W., Model-Checking for Real-Time Systems. In *Proceedings of the 10th International Conference on Fundamentals of Computation Theory*, Dresden, Germany, 22-25 August, 1995.

[31] Larsen, G., Pettersson, P., Yi, W., “UPPAAL: Status & Developments,” In *Proceedings of the 1997 Computer-Aided Verification*, CAV’97, Israel, June 1997, Lecture Notes in Computer Science, Springer-Verlag, 1997.

[32] Larsen, K., Pettersson, P., Yi, W., Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, Pisa, Italy, 5-7 December, 1995.

[33] Maler, O., Yovine, S., “Hardare Timing Verification using KRONOS,” In *Proceedings of the IEEE 7th Israeli Conference on Computer Systems and Software Engineering, ICCBSSE'96*, Herzliya, Israel, June 12-13, 1996.

[34] Bouajjani, A., Tripakis, A., and Yovine, S., “On-the-fly symbolic model-checking for real-time systems,” In *Proceedings of the 1997 IEEE Real-Time Systems Symposium, RTSS'97*, San Francisco, CA, USA, December 1997.

[35] Daws, C., Yovine, S., “Reducing the number of clock variables of timed automata,” In *Proceedings of the 1996 IEEE Real-Time Systems Symposium*, Washington, DC, December, 1996.

[36] Daws, C., “Optikron: A Tool Suite for Enhancing Model-Checking of Real-Time Systems,” In *Proceedings of the 1998 Computer-Aided Verification*, CAV’98, Israel, June 1998, Lecture Notes in Computer Science, Springer-Verlag, 1998.

[37] Lee, E., “Embedded Software,” *Advances in computers*, 56, 2002.

[38] Harel, D., et al., “STATEMATE: A Working Environment for the Development of Complex Reactive Systems,” *IEEE Transactions on Software Engineering*, Vol. 16., No. 4, April 1990, pages 403-413.

[39] Harel, D., Naamad, A., “The STATEMATE Semantics of Statecharts,” *ACM Transactions on Software Engineering Methods*, Vol. 5, No. 4, October 1996.

[40] UML Version 1.3 Specification.

[41] M. von der Beeck "A Comparison of Statechart Variants" In Formal Techniques in Real-Time and Fault-Tolerant Systems, number 863, in Lecture Notes in Computer Science, 1993.

[42] Rajeev Alur, David Dill, "A Theory of Timed Automata," *Theoretical Computer Science* Vol. 126, No. 2, pp. 183-235 1994.

[43] Lee, E., Parks, T., "Dataflow Process Networks", *Proceedings of the IEEE*, pp. 773-799, May 1995.

[44] Lee, E., Messerschmitt, D. "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, Vol. 36, No. 1, pp. 24-35, 1987.

[45] Parks, T., "Bounded Scheduling of Process Networks," Ph.D. dissertation, Technical Report UCB/ERL-95-105, EECS Department, University of California, December, 1995.

[46] Ranjan, R., "Design and Implementation Verification of Finite State Systems," PhD thesis, University of California, Berkeley, 1997.

[47] Helbig, J., Kelb, P., "An OBDD Representation of Statecharts", *Proceedings of the European Conference on Design Automation*, pages 142-151, Paris, France, 1994.

[48] Burch, J., Clarke, E., Long, D., "Symbolic Model Checking with Partitioned Transition Relations", October 1991, CMS-CS-91-195.

[49] Chan, W., Anderson, R., Beame, P., Jones, D., Notkin, D., Warner, W., "Optimizing Symbolic Model Checking for Statecharts", *IEEE Transactions on Software Engineering*, Vol. 27, No. 1, pp. 170-190, 2001.

[50] Bhattacharyya, S., Praveen K. Murthy, P., Lee, E., "Synthesis of Embedded Software from Synchronous Dataflow Specifications," *Journal of VLSI Signal Processing*, vol 21, pages 151–166, 1999.

[51] Bapty T., Neema S., Scott J., Sztipanovits J., Asaad S.: "Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems," *VLSI Design*, 10, 3, pp. 281-306, 2000.

[52] Scott J., Bapty T., Neema S.: "Runtime Environment for Dynamically Reconfigurable Embedded Systems," *Proceedings of the International Conference on Signal Processing Applications and Technology*, CD-ROM, Orlando, FL, November, 1999.

[53] Edwards, S., "The Specification and Execution of Heterogeneous Synchronous Reactive Systems," PhD Thesis, University of California, Berkeley, 1997.

[54] Murthy, P. "Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow," Memorandum No. UCB/ERL M96/79, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Ca 94720, December 10, 1996.

[55] Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., McKenzie, P., Systems and Software Verification, Springer, 2001.

[56] Heitmeyer, C., Mandrioli, D., Formal Methods for Real-Time Computing, John Wiley & Sons, 1996.

[57] Davis, J., Scott, J., Sztipanovits, J., Karsai, G., Martinez, M.: "An Integrated Multi-Domain Analysis Environment For High Consequence Systems", *Proceedings of the 1998 ASME Design Engineering Technical Conference / Computers in Engineering*, Atlanta, GA, September 1998.

[58] Davis, J., Scott, J., Sztipanovits, J., Karsai, G., Martinez, M., "Integrated Analysis Environment for High Impact Systems", *Engineering of Computer Based Systems*, Jerusalem, Israel, April 1998.

[59] Davis, J., "Intergrated Safety, Reliability, and Diagnostics of High Assurance, High Consequence Systems," Dissertation, Vanderbilt University, 2000.

[60] Ruf, J., Kropf, T., "Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals", Tech. Rep. SFB358-C2-1/97, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, April 1997.

[61] Havelund, K., Skou, A., Larsen, K., Lund, K., "Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL," BRICS Technical Report RS-97-31, November 1997.