Formal Verification of the Gigamax Cache Consistency Protocol

K. L. McMillan
Carnegie Mellon University
Pittsburgh, PA 15213
mcmillan@cs.cmu.edu

J. Schwalbe
Encore Computer Corporation
Marlborough, MA 01752-3004
schwalbe@encore.encore.com

February 18, 1997

Abstract

We have been using *symbolic model checking* techniques to verify the cache consistency protocols of the Encore Gigimax multiprocessor. These are exceedingly complex protocols designed to ensure consistency at the individual cache line level between caches residing on separate backplanes, and use both "snoopy" cache techniques, and message passing between backplanes. Automatic analysis of our formal model uncovered execution traces leading to failures which had not been found in the design phase or in simulation. Symbolic model checking techniques made it possible to search exhaustively the very large state space of the model. Although this technique has shown great promise in early experiments, this is the first time it has been applied to a real industrial design. The paper covers the abstraction methods by which the formal model was constructed, an introduction to symbolic model checking methods, our experience constructing and analyzing the Gigamax protocol model, and a few practical remarks about how this kind of formal analysis fits into the design cycle. In addition, there are a few words about future plans for the Gigamax experiment, including the application of structural induction methods to the model, and the successive refinement of the model to levels of greater detail.

1 Introduction

This paper describes the application of an automatic formal verification method called symbolic model checking to a complex, real-life hardware system. The system is the memory hierarchy of the Gigamax multiprocessor, under development at Encore Computer Corporation. The complexity of this system, which uses both snooping cache methods and a message passing protocol to maintain consistency of distributed caches, led the designers to consider automatic verification as a debugging tool, since it wasn't clear that traditional simulation methods could provide a sufficient degree of confidence in the system's correctness. After studying the system architecture, we constructed a formal model of the system at a level of abstraction which we felt would provide useful information about the design's correctness, without overwhelming detail. This model was then analyzed using a relatively new technique called symbolic model checking. A symbolic model checker performs an exhaustive search of the model's state space without explicitly constructing the global state graph. Instead, it represents the value of predicates over the state space in symbolic form, in this case using the canonical "Boolean decision diagram" form. Checking the model for two important properties of a distributed memory system (consistency and absence of deadlock) exposed a number of design errors. As the design evolved to correct these errors, the model was easily adapted, and was able to quickly provide an analysis of any new errors introduced by design changes. It is hoped that the results of this experiment will provide an additional argument for constructing abstract formal models of complex protocols before proceeding with detailed design.

The first part of this paper sketches the symbolic model checking method of automatic verification. A more detailed description of the method is given in [BCM+90, BCMD90]. Section 2 provides an overview of the Gigamax memory architecture. Section 3 describes the construction of the formal model and the principle of abstraction on which it is based. The formal specification is then discussed in section 4, and some of the analysis results are described in section 5. Particular attention is given to relating the aspects of the model which result in efficient performance of the symbolic model checker. These aspects of the model would seem to be fairly general and common to many computer system designs. Finally, since the verification of the Gigamax protocols is ongoing research, we describe the directions which we expect the research to take in section 6. For techniques related to symbolic model checking see [Bry88, CBM89, BF89a, BF89b].

2 Symbolic Model Checking

A model checker is a program which can decide whether a given finite model satisfies a formula in a logic. For example, given the transition structure of a finite state computation, a model checker can determine the truth of a formula in branching time temporal logic (CTL) in time which is linear in the size of the model and the size of the formula [CES86]. CTL is a propositional logic, with additional operators for expressing temporal relationships. For example, if g is a formula, then $\mathbf{F}g$ is true if g holds at some time in the future. The path quantified formula $\forall\mathbf{F}g$ holds if g holds at some time in all possible futures, while $\exists\mathbf{F}g$ holds if g holds at some time in some possible future. A typical model checking program for CTL first constructs a complete state graph of the system, then uses efficient algorithms for graph reachability and strongly connected components to label the states satisfying each subformula of the specification. Examples of finite state systems which have been verified, or partially verified, using this type of CTL model checker can be found in [CBBG87, CLM89].

A symbolic model checker differs from an explicit model checker in its representation of structures. Instead of representing a set of states by labeling the global state graph, we represent it with a logical formula which is satisfied in a given state if and only if the state is a member of the set. Consider, for example, a distributed system in which each component of the system holds a copy of a data object. A protocol insures that all valid copies are consistent, and at least one copy is valid. A simple expression in first order logic which defines the set of reachable states of the system is

$$\forall i,j[\mathit{Valid}(i) \wedge \mathit{Valid}(j) \rightarrow (\mathit{value}(i) = \mathit{value}(j))] \wedge \exists i[\mathit{valid}(i)]$$

Here, we have defined the state in terms of a unary predicate *Valid* on components, and a unary function *value* from components to data values. If the formula is interpreted over a set of components of cardinality n, and a set of data values of cardinality m, then the formula concisely represents a set of $O(m2^n)$ states.

This is not to suggest that first order finitary logic would be a reasonable choice for a symbolic representation in automatic verification; only that symbolic forms can be much more compact than enumeration. The first order formalism has a number of useful properties, however, from a theoretical point of view. For example, we can perform the usual set theoretic operations directly on this representation, using disjunction for union and negation for complementation. The transition relation of a system can be represented by a formula τ of two free variables x and y, which is true if and only if $\langle x, y\rangle$ is a transition. If we represent the set of initial states of a system by a formula f of one free variable x, then the parametric form $\lambda y[\exists x[\tau \wedge f]](x)$ represents the set of states reachable in one transition. We can read this formula as representing the set of all states y such that there exists a state x such that $\langle x, y\rangle$ is a valid transition and x is initial. Thus we can compute a representation of the set of states reachable in one transition by a purely syntactic transformation on formulas. We can also inductively characterize useful sets such as the set of states reachable in any number of transitions from an initial set, or the set of states satisfying a formula in CTL, using the Mu-Calculus fixed point notation [BCM+90]. For example, the set of states reachable in any number of transitions from the initial state is expressed by the Mu-Calclulus formula $\mu S[f \vee \lambda y[\exists x[\tau \wedge S]](x)]$. This fixed point expression expands to a series of formulas representing the state sets reachable after $0, 1, 2, \ldots$ transitions from the initial states. Thus, the fixed point expansion is effectively a symbolic breadth first expansion of the state space. The series of formulas represented by the fixed point expression is guaranteed to converge if the ranges of the variables are finite, and if the fixed point parameter S only appears under an even number of negations. Since detecting this convergence depends on being able to decide the equivalence of two consecutive formulas in the series, it is desirable (though not absolutely necessary) to be able to keep formulas in a canonical form.

In practice, we can apply various canonical forms for propositional formulas, such as Boolean Decision Diagrams (BDD's) [Bry86] or Typed Decision Graphs (TDG's) [Bil87]. Both are forms of binary decision trees, in which variables are restricted to appear in a fixed order. Heuristically, this ordering results in many occurrences of identical subtrees, which can be shared in the representation, giving the formula a DAG (directed acyclic graph) structure. This sharing of subformulas is particularly useful in representing the reachable state set of loosely coupled systems, where each component of the system has only weak knowledge of the state of the rest of the system. There are efficient algorithms for reducing formulas of QBF, essentially first order logic over the set $\{0, 1\}$, into the BDD or TDG canonical forms. In this way, we can build, for example, a CTL model checker which operates entirely on symbolic forms. The efficiency of a symbolic model checker depends on how concisely these canonical forms represent the transition relation and the series of formulas resulting from fixed point expansions, in particular, the set of reachable states of the system. For example, consider the construction of a BDD for the reachable state set of the distributed data object described above. At each level of the DAG structured decision tree, there are only $m+1$ subtrees, representing the m different possible data values, plus the possibility that no component at or below that level is valid. Thus the size of the BDD representing the reachable state set is $O(nm)$.

One final point worth mentioning about the current implementation of the symbolic model checker is that, since BDD's contain only Boolean variables, the state of any system to be verified must be encoded in binary. This is not a limitation in principle, especially for hardware designs, since they are necessarily encoded in binary in order to be implemented in a digital technology. In behavioral level models, however, it is useful to have a broader range of data types, since the binary encoding of the system state is not relevant at this level, and only adds spurious detail. It is reasonable to expect, however, that a model checker based on structures similar to BDD's but over multi-valued variables would allow the model to be expressed more concisely at a high level, but provide a similar efficiency in representation.

3 The Gigamax Memory Architecture

The Gigamax is a distributed, shared memory multiprocessor under development for DARPA at Encore Computer Corporation. The gross architecture of the Gigamax memory system is depicted in figure 1. The system is organized into a network of processor *clusters*. Each cluster is an Encore Multimax with

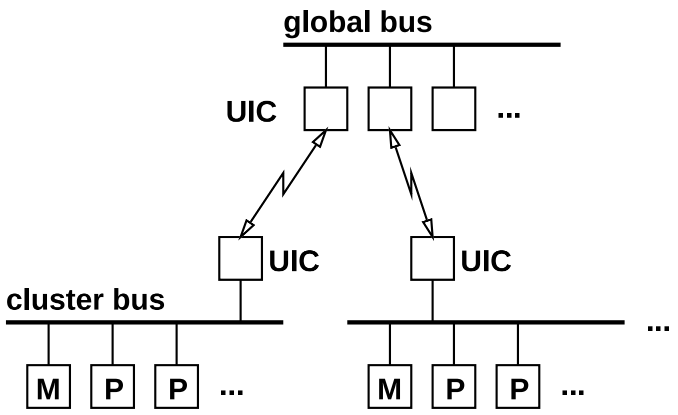


Figure 1: Gigamax memory architecture

a high speed, split transaction bus (called a *nanobus*), hosting a number of processor boards, some main memory, and a special interface card called a *UIC*. The UIC links the cluster into a star network. This network is centered around a global cluster, housing a collection of UIC's which link the global bus to each processor cluster. The processor boards within one cluster use a *snooping cache* protocol, similar to the protocols described in [AB86], but adapted for a split transaction bus. When a processor encounters a read (write) miss in its local cache, it issues a *read public* (*read private*) command on the nanobus, along with a tag indicating the source of the command. It then frees the bus for other transactions, while waiting for the memory, which is pipelined, to respond. The response to a read command contains the tag as a destination, and may be overlapped with any bus cycle which doesn't use the data part of the bus.

Each cache has a *bus watcher*, which keeps a copy of the cache tags. When the bus watcher detects an address on the bus which hits in the local cache, it may intervene by asserting the *memory bypass* signal on the bus and/or sending a command to the local cache to flush or invalidate the line. The specific response depends on the type of bus command and the *state* stored in the tag memory. Assertion of the memory bypass signal prevents the memory from responding to the command, and indicates that the asserting cache will respond to the command in the future with a *write response* command, which has the effect of simultaneously updating the contents of the main memory, and supplying the data to the original requester.

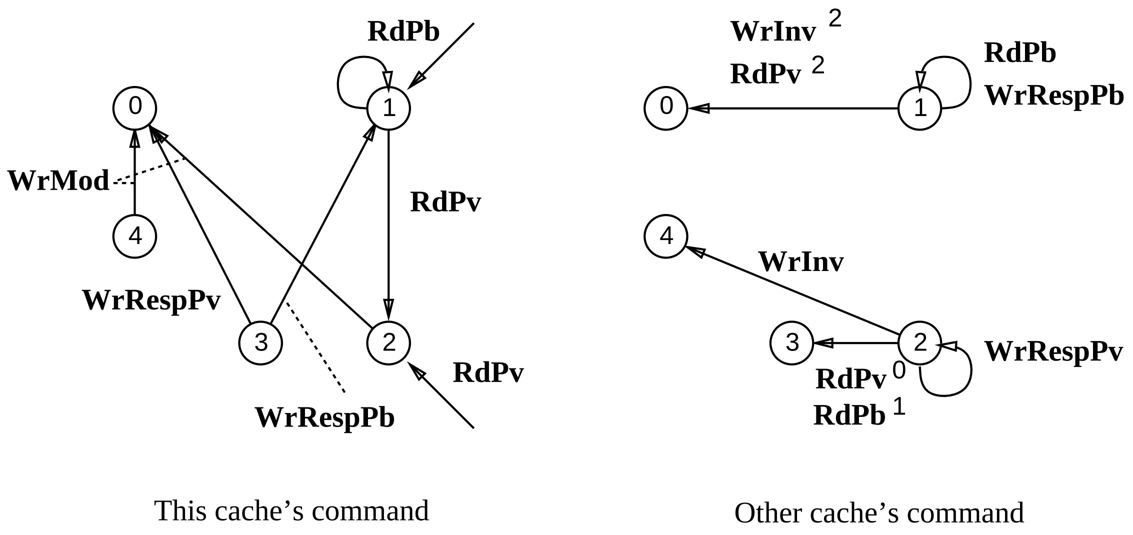
A simplified bus watcher state diagram is given in figure 2. The bus commands not mentioned previously are *write modified*, which occurs when a modified line is replaced in the cache, and *write invalidate*, which is used by non-caching devices and as an invalidation signal from remote clusters. Of particular interest are transitions to and from the *owned* state. A bus watcher enters the *owned* state when it's cache issues a *read private* command. Any other bus watcher in the owned state at this time asserts *memory bypass*, sends a *flush private* command to its local cache, and enters a special transitional state called *Xown*. Later, this cache will issue a *write response private* command on the bus, at which time the requester's cache goes valid, and the responder's bus watcher goes to the *invalid* state. In order to prevent any other requesters from interfering with this two phase ownership transfer process, an interlock is set in the requesting cache's bus watcher, which causes all future accesses to the address in question to be stalled until the ownership transfer is complete.

Each UIC also has a bus watcher which keeps track of addresses from the local main memory which are checked out in remote clusters. This allows the UIC to intervene on the bus on behalf of caches in remote clusters, eliminating the need for all processors to share the same bus. This system differs from a directory based cache system in that the UIC does not store the identities of all the other caches holding checked out copies. Instead, the tags stored in the global bus UIC's allow *invalidate*, *flush public* and *flush private* commands to be routed to all appropriate clusters. The UIC decides which command, if any, should be forwarded to the link based on it's bus watcher state, the bus command type, whether or not the address is local, and the response of other bus watchers to the command. The global bus UIC's are identical to the cluster UIC's, except in their address decoding.

As an example if the Gigamax operation, consider the case where a miss occurs on a read in a local processor cache and where the address is located in a remote memory. This causes the local cache to issue a *read public* command on the bus. An interlock circuit is set to cancel any future access to that address while this command is pending. Now let's consider two cases:

- There is a cache in the local cluster in the *owned* state. The bus watcher of this cache intervenes in the transaction by asserting *memory bypass*. The requesting cache's bus watcher enters the *shareded* state, while the intervening bus watcher enters the transitional state *Xown*, sending a *flush public* command to it's local cache. Later, this cache issues a *write response public* command which causes its bus watchers to transition from the *Xown* state to the *shared* state, and the interlock to be canceled. The UIC forwards this *write response public* command to the link with a null destination field, which causes the data to be written back to the remote main memory, since memory is now the implicit owner.

- No other local cache has a valid copy, but a cache in a different cluster is in the *owned* state. The local UIC forwards the *read public* command to the link (its default behavior for remote addresses). The command is issued on the global bus, setting an interlock on the global bus for this address. The global bus UIC connected to the cluster with the *owned* copy intervenes, forwarding the *read public* command to this cluster, and entering the *Xown* state. Eventually, a *write response public* returns from the remote cluster,



State 0 : Invalid
State 1 : Shared
State 2 : Owned
State 3 : Xown
State 4 : Xmem

Notes:
0 : memory bypass / flush private
1 : memory bypass / flush public
2 : invalidate

Figure 2: Bus watcher state diagram

its global UIC goes to the *shared* state, the *write response public* is returned to the local cluster, and the interlock is canceled on the global bus. Since main memory again becomes the implicit owner, the *write response public* is also forwarded to the main memory.

These examples must serve to illustrate the protocol, since the complete set of rules governing the bus watchers, interlocks, and command forwarding is too complex to enumerate here. The interactions described above are fairly straightforward cases, but a number of unexpected things can happen when requests for the same location occur simultaneously on different clusters. When this happens, local state changes occur which are not immediately reflected in the state of remote UIC's, because of the latency in communication. This leads to highly complex interactions between clusters, and as we will see, can lead to deadlock.

4 Constructing the Formal Model

There are a number of issues involved in constructing a formal model for automatic verification. The most important of these is choosing the right level of abstraction. The process of abstraction allows us to construct a model which is manageable in complexity. A model, in the temporal logic framework, is called a *Kripke structure*. The Kripke structure is a directed graph. The nodes of this graph are states, each of which is labeled with the truth values of some propositions about the world, and the arcs are transitions which define the possible sequences of states (computations) which might be observed. Formal abstraction means, in a sense, throwing away some information about the state of the system in order to simplify the model. In the process, some states become indistinguishable. Thus each state in an abstract model may correspond to several states in a more detailed or refined model (see figure 3). In order to make the abstract model a sound basis for reasoning about the system, it is constructed conservatively: if insufficient information exists in the abstract model to determine whether a transition is possible, one assumes that the transition might occur. Thus, throwing away information about the system state results in a less deterministic model, such that each computation in the detailed model corresponds to a computation on the abstract model. Because of this conservative construction, the abstract model preserves a certain class of temporal properties called *safety* properties. The application of this kind of abstraction, formally a *homomorphism* between the two structures, has been extensively studied by Kurshan [Kur86]. Using ω-automata as the basic structures, it is possible to construct homomorphisms which preserve a more general class of properties, called ω-regular properties. These properties include those characterized as liveness properties, and all properties expressible in linear temporal logic (LTL).

Because model checking techniques are in some measure limited by the number of states of the model, and cache memory system has a very large amount

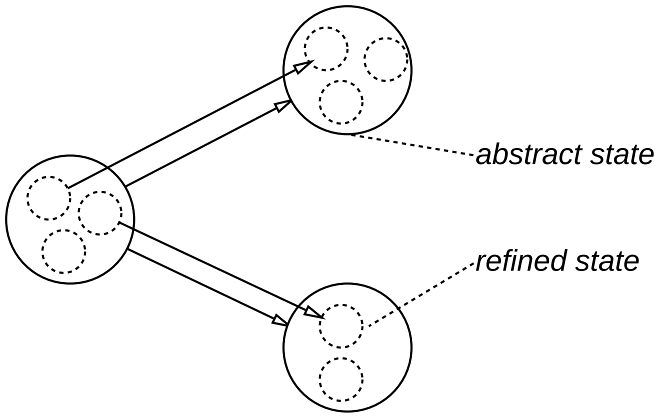


Figure 3: Abstracting state information.

of state, some kind of abstraction is necessary.[1] The first abstraction we make in modeling the system is to focus on a single memory address in the system, and throw away any information about other addresses. This abstraction will, of course have an impact on the kinds of questions we can answer about the system. For example, with no information about transactions on other memory locations, it is not possible to determine when a given address is replaced in a given cache. It must therefore be assumed that replacement may occur at any time. While this may prevent us from proving certain properties of the system, it has the advantage that properties we do prove hold independently of the cache replacement policy. Thus, by taking a bird's eye view, we can prove properties of a general class of systems rather than a particular implementation. As we will see, some useful properties of the system can be proved by considering only the information associated with one memory address. These results may be generalized to the other memory addresses by symmetry arguments. In other cases, bugs in the design may be found automatically, even though not enough information exists in the model to prove that the system is correct.

The abstract model of the Gigamax architecture is depicted in figure 4. The cluster model contains two processors, a local memory, and the UIC. The global bus model contains a UIC linking it to one cluster model, and a set of abstract UIC/cluster models, which represent the combination of a UIC card connected to a remote cluster. Thus, different components in the model are represented at different levels of detail. Each component of the model maintains only the state information relating to the address under analysis. For example, the processor board components store the bus watcher state of that address (invalid, shared, owned, *etc.*), one bit of the data held in the cache, and the state of the interlock

[1] In fact, viewing any physical system as a discrete, or digital, system is a kind of abstraction, though it is such a common abstraction that it is often thought of as reality.

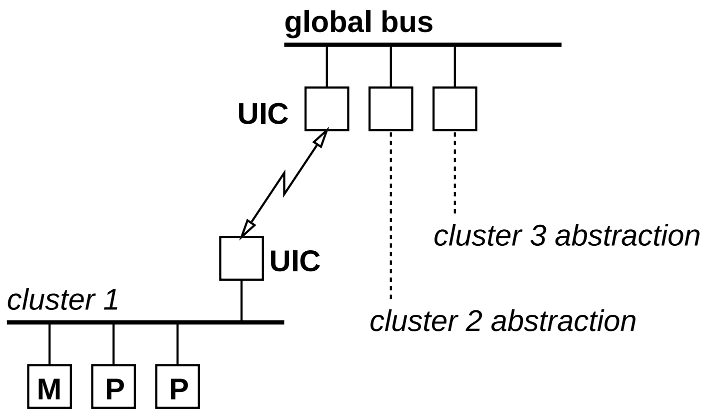


Figure 4: An unbalanced abstraction

system *vis-a-vis* that address. It also stores some information about commands to the local cache that are pending. The UIC components record similar state information, but also keep track of the messages enqueued in the communication link. The link maintains two queues of messages traveling in each direction. One of these queues holds *read private* and *read public* commands, while the other queue holds all other types of commands. Since the model only contains information about messages associated with the address in question, the number of entries in a queue at any given time is not known. This introduces some uncertainty into the transit time of messages in the system.

4.1 Form of the model

The input format for the symbolic model checker is actually a complete mini-HDL, with several useful theoretical properties which are not present in most current HDL's. Among these are a simple semantics based on first order logic, the ability to underspecify a system (*ie.*, non-determinism), and a notion of module composition which allows the use of formal refinement techniques. The transition relation of the model is specified in first order logic in terms of the current and next values of the state variables, and optionally a collection of auxiliary variables, which can be used to represent the value of communication signals. The auxiliary variables are viewed as being implicitly existentially quantified. A valid transition is an assignment to the current state and next state variables which satisfies the state transition formula. Composition of modules is accomplished by the logical conjunction of transition formulas. Therefore, the logical properties of conjunction can be used in formal decompositions of the model. Breaking a model into components and then formally refining the component models is a technique which is of primary importance in managing the complexity of concurrent systems [Kur86]. This technique is justified in the case of our mini-HDL by a simple, but quite useful property of conjunction: that $a \rightarrow a'$ and $b \rightarrow b'$ implies $(a \wedge b) \rightarrow (a' \wedge b')$. This *monotonicity* property allows components of a system to be refined separately while preserving certain classes of properties of the composition. It also allows a simple induction technique over unbounded arrays of processes [KM89].

The mini-HDL also has a highly general facility for defining and instantiating module types, with substitution of parameters, generation of hierarchical names, *etc.* It also adds a thin layer of syntactic sugar over the underlying logical notation. Thus the **case** notation of figure 5 is used to describe a list if rules to be used in prioritized order for transitions of a bus watcher from the *owned* state. This notation is translated into logic using implication and conjunction, but the case list form is much more readable. An additional feature of the language allows the description of systems which have components operating on separate clocks, such as the separate clusters in the Gigamax model. The transitions of these components are interleaved arbitrarily. This interleaving behavior can be modeled by taking the logical disjunction of the transition relations. Empirically, however, it was found more efficient to represent these relations with separate BDD's and to form instead the logical disjunction of the sets reached via each relation.

5 The Formal Specifications

Using the symbolic model checking algorithm, we can verify a variety of properties of the abstract model. The first part of the specification of the Gigamax memory system has to do with the consistency of entries for the same address in different caches. Since the Gigamax implements a weakly consistent shared memory, the basic safety specification of the system is really a simple model of a weakly consistent shared memory, as depicted in figure 6. In this model, all writes occur to a single copy of the data, but reads are directed to a distributed set of copies, which are independently updated from time to time from the master copy. Thus a cache with a shared copy may hold stale data, at least for a short amount of time. Note that this abstract model is an incomplete specification, since it does not detail the precise timing of update events. In order to verify that the Gigamax model correctly implements it, we effectively run both models in parallel and compare the outputs. Because the specification is incomplete, we drive the arbitrary choices in the specification from the implementation. In this way, we prove a refinement relation between implementation and specification rather than equivalence. The proof is by symbolic breadth-first expansion of the state space of this parallel composition, using the symbolic model checker. If at any state the outputs of the specification and implementation fail to correspond, the proof fails and the search is traced back by the model checker to produce a counterexample. Note that in this case, we are

```
case
 ...
  Owned: case
    Stall | Idle : next(Owned);
    !mycyc : case
      WrInv & !RC : next(Xmem1);
      EchoRespPv & !RC : next(Owned);
      WrRespPv & RC : next(Owned);
      RdResp & RC: next(Owned);
      WrRespPbAlt & RC : next(Owned);
      WrMod & RC : next(Owned);
      CNack : next(Owned);
      RdPb : next(Xown1 & !FlPvPend);
      RdPv : next(Xown1 & FlPvPend);
      else: next(errstate);
      esac;
    else : case
      WrMod & !RC : next(Invalid);
      WrRespPbAlt & !RC : next(Shared);
      else : next(errstate);
      esac;
    esac;
  ...
esac
```

Figure 5: Fragment of Gigamax model text

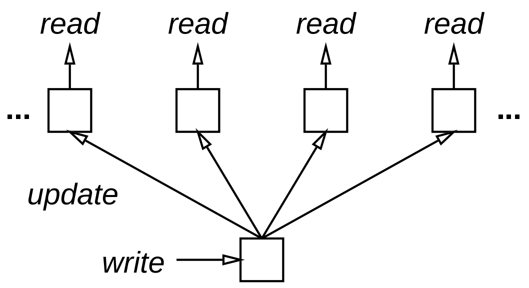


Figure 6: Model of weakly consistent shared memory

using a CTL model checker as a tool for testing a relationship between two finite models, a technique which is often more convenient that specification directly in CTL. The weak consistency property was the first one that we succeeded in proving.

Another safety property we wanted to check related to the internal diagnostics in the Gigamax protocol. Each bus watcher in the system signals an error when it sees an unexpected event on the bus. We wanted to verify that an error could never be signaled during correct operation of the system. This proved to be a particularly useful application of the model checker which turned up numerous bugs in the design, since predicting by hand which bus events might be seen by a bus watcher in any given state turned out to be an extremely error prone process. In fact, it would have been possible for the model checker to compute the conditions for signaling errors, although we had no tools for directly translating this into the design.

Absence of deadlock is another important property for which we wished to test the model. A deadlock occurs when one or more processors becomes locked out from reaching the *shared* or *owned* state. For each bus watcher in the model, we specify

$$\forall \mathbf{G}[\exists \mathbf{F}\, owned \wedge \exists \mathbf{F}\, shared].$$

In other words, from every reachable state, there exist computation paths leading to states in which the given bus watcher is in the *owned* and *shared* state respectively. We refer to the absence of such a path as a deadlock. Because this properties relies on the *existence* of computation paths, it isn't necessarily preserved by refinement. For example, since our abstract model of the Gigamax has no state information relating to the scheduling of the bus, or the precise response times of certain components, certain potential livelock conditions will be ignored. From a practical point of view, however, the abstract model allowed us to find two significant deadlocks in the protocol (one of which was not found in simulation, and is described in the next section). Subsequent simulation revealed additional deterministic livelocks using a detailed model of the communications links.

Ideally, we would like to show that all requests are eventually served. Unfortunately, because of the nature of the Gigamax bus protocols, this property only holds in a probabilistic sense. In fact, the system has watchdog timers to signal an interrupt in the very rare case when one processor is locked out for an extended period of time. Since we have no formal machinery for proving probabilistic properties of finite state systems, we concentrated on proving that no processor is ever deterministically locked out.

6 Results of the Analysis

We now examine the performance of the symbolic model checking algorithm in verifying the Gigamax model with respect to our specifications. The model checker first constructs a BDD representation of the transition relation(s) of the model. (Recall that the model's behavior is an arbitrary interleaving of the transitions of its asynchronous components.) The reached state set is then computed by the fixed point expansion described in section 2 and is also represented by a BDD. The CTL formulas in the specification are then evaluated. This uses an algorithm which is similar to the fixed point expansion of the reachable state space, but searches backward instead of forward. Thus, to find all of the states which satisfy $\exists \mathbf{F} g$, we begin with the set of states satisfying g, then search backward in a breadth first way to find all of the states which can reach a state satisfying g. This search is constrained within the set of states reachable from the initial state, as this has been found empirically to reduce the size of the BDD's involved.

Table 1 summarizes three separate runs of the model checker. The first run was to check the absence of deadlock property on a model with the data part completed abstracted. The second run checked the consistency property, and failed at a search depth of 5. The third run used a later model with a second cluster expanded in detail, checking only safety properties. The number of states here is comparable to that in the first run because of a simplification of the model of the message queues. The table gives the largest number of BDD nodes needed represent a transition relation (TR) and the reached state set (RS), the total number of states searched (SS), and the depth of the search (DS). A comparison of the number of reached states with the size of the BDD's is of particular interest, since traditional model checking algorithms must store the entire global state graph. With state spaces as large as 10^{13} states, it is clear that the models could not have been handled directly using traditional model checking methods. None of these three runs required more than a few minutes, however, using a symbolic model checker running on a Sun3/60.

Later, we will examine the sources of the symbolic model checker's efficiency for this problem. For now, though, we consider a counterexample of the absence of deadlock specification which was discovered by the model checker. With

run	TR	RS	SS	DS
1	11146	1429	2.99×10^9	15
2	44100	898	1.04×10^{13}	5
3	8823	2550	3.30×10^9	20

Table 1: Quantitative results of symbolic model checking

reference to figure 4, the counterexample begins with a processor in cluster 2 holding an *owned* copy of an address checked out from the main memory of cluster 1. It proceeds in the following steps.

1. A processor in cluster 1 encounters a read miss, and issues a *read public* command. The UIC in cluster 1 intervenes in this request and sends a *flush public* command (implemented by *read public*) to the global bus, which stores the command in a queue. The processor in cluster 1 sets its interlock.

2. A processor in cluster 3 encounters a read miss, and issues a *read public* command. This command reaches the global bus, where the global UIC for cluster 2 intervenes (since cluster 2 holds an owned copy), and a *flush public* is sent to cluster 2.

3. The response from cluster 2 (a *write response public* command) is transmitted via the global bus to cluster 3. This results in the global UIC bus watchers for both clusters being in the shared state. Since main memory is now implicit owner, a *write response public* command, with null destination, is sent to cluster 1 by the global UIC for cluster 1.

4. The *write response public* message reaches cluster 1, and the data are deposited in main memory.

5. The processor in cluster 2 invalidates its copy by cache replacement, then encounters a read miss and issues a *read public* command, which is forwarded to the global bus. Since no global UIC is in the *owned* state, the command is routed normally to cluster 1 and enqueued there. The global bus UIC for cluster 2 sets its interlock.

6. The original *read public* command from step 1 is issued on the global bus, but is blocked by the interlock held by cluster 2. Likewise the *read public* from step 5 is issued on the cluster 1 bus, and is blocked by the interlock set in step 1. Since neither interlock can be released until the request that switched it on is satisfied, a deadlock occurs at this point.

This is an example of the classic deadlock situation which occurs when two processes attempt to obtain locks on two resources in different orders. Nonetheless, the sequence of events that lead to this situation were sufficiently complex that the designers were unable to predict that the situation might occur. In fact, the deadlock situation was found at a search depth of thirteen transitions. At each step in this sequence, there were several alternatives that might have averted the deadlock. Thus it is unlikely that this deadlock would be found by a random simulation run, or a simulation run based on address traces.[2]

The fact the the model checker was able to print out automatically an example of this deadlock highlights an important difference between a model checking and a theorem proving approach to verification. In a theorem proving approach, a failure to prove the system correct indicates either that the system is incorrect, or that insufficient resources were dedicated to the theorem proving process (whether it is automatic, or only partially automatic). In the case of model checking, if the system is incorrect, the model checker will produce a counterexample demonstrating incorrect behavior. These counterexamples are of perhaps even greater value than a proof that the system is correct, since such a proof is useful only to the level of detail at which the system is modeled, and only if the original specification was correct and complete.

Some explanation is in order for the remarkable efficiency of BDD representation used in the symbolic model checker. The efficiency in representing the transition relation derives from the bounded amount of communication which occurs over the system busses in each system cycle. In fact, the BDD representation of the transition relation of any finite state bus based system (including "wired or" busses) is linear space in the number of bus clients. This can be shown using automata theoretic arguments which are a bit too detailed to present here. As to the efficiency of representing the set of reached states, we return to the statement made in section 2 that BDD's represent efficiently the set of reachable states of loosely coupled systems, where each component of the system has only weak knowledge of the state of the rest of the system.

We can quantify this notion using a quasi information theoretic measure. We begin by partitioning the set of state variables of the model into components of bounded size, according to the modular structure of the system. This partitioning defines, in some sense, our notion of locality in the system. Our definition of "loosely coupled", is relative to this notion of locality, or granularity. Roughly speaking, it measures how much we know about the state of some subset of the components, given the state of the remaining components. Now let A be the union of some subset of these components. Given a value assignment $V : A \rightarrow \mathcal{D}$ to the variables in A, let $R|V$ be the set of reachable states

[2]In fact, the number of possible transitions from a given state ranges from 6 to 12. The probability of a random simulation run executing this trace is therefore in the range $6^{-13} = 7.7 \times 10^{-11}$ to $12^{-13} = 9.3 \times 10^{-15}$. The expected time for a random simulation to exhibit this behavior would be somewhere between 2.4 years and 29 millenia, assuming the simulation could be carried out at 10,000 steps per second.

of the system which are consistent with V. We define $K(A)$ to be the number of *distinct* such subsets, for any V. We conjecture that in the Gigamax model, $K(A)$ is bounded by a small constant, for any subset A of components. This is plausible because the possible states of the components not in A depend only on a few conditions on the state of A, such as whether or not it contains any processor in the *owned* state, or has any interlock set, or has a certain command in some queue. If $\max_A K(A)$ is small compared to the number of reachable states of the system, we say the system is loosely coupled. The size of the BDD representation of a set is bounded by $O(n \max_A K(A))$, where n is the number of variables in the system. Thus, the reachable state set of a loosely coupled system is compactly represented by a BDD.

7 Future Plans and Conclusions

There are a number of limitations to the current work, which we will attempt to address in the future. One obvious area of extension is in refining the model to include more detail of the protocol implementation. A refined model, for example, could reveal deterministic livelocks which are not apparent in the high level model. Although verification at the architectural level has been useful, this has not prevented significant problems from developing in aspects of the design below this level of detail. Once the high level protocol is verified, it is important to verify that the individual components of the system are implemented correctly. Perhaps a more important problem is that the current model considers only one memory address in the system. By including information about two or more memory locations that may replace each other in the caches, more information may be obtained about deadlocks relating to cache replacement. (Recall that in the model, replacement may occur at any time. In actuality, replacement will only occur when all associates are full. This more restricted behavior may not lead to any consistency violations not present in the more general model, but may lead, for example, to livelocks which would not otherwise occur). Also, there are other properties of the model that should be verified if more than one memory location is modeled. For example, the system guarantees write sequentiality under certain circumstances, and this should be formally verified.

Another possible area of extension is proving that the system specifications are satisfied for models with an arbitrary rather than a fixed number of bus clients. This can in theory be accomplished using the induction method described in [KM89]. An example of this process can already be seen in the abstract UIC/cluster model. If this model can be shown to be a formal abstraction of a UIC model connected to a cluster bus which in turn is a collection of UIC/cluster models, then an inductive argument will show that it is an abstraction of a hierarchy of any depth. The UIC/cluster model serves in this case as a *process invariant*. Finding such invariants is the key to doing inductive proofs about processes. To show that the system specification holds for a cluster bus with an arbitrary number of clients, it is sufficient to find a process invariant which is an abstraction of itself with one added bus client, and which itself satisfies the specification. This method is practical if such an invariant can be constructed for realisticly complex models, such as the present one. At present, only simple, fairly theoretical problems have been solved using this method, and it will be interesting to see how well it transfers to practice.

What conclusions can we draw from this experience? It seems to support the notion that formal modeling can be a valuable tool in the early phases of design of a complex protocol, and can save a great deal of wasted effort in the detailed design of systems which are flawed at the architectural level. The techniques presented here are limited to systems which can be represented with finite state models. This would seem to cover a fairly broad range of protocols implemented in both hardware and software, however. Another apparent conclusion is that the number of states of a system is not a very good measure of the complexity of verifying the system automatically, and that other measures based on the structure of the state space may be more appropriate. Using techniques of abstraction and representations which are suitable to the problem space, such as BDD's, we can automatically prove properties of extremely complex systems. Perhaps more importantly, automatic verification systems can produce examples of behavior which contradicts the specification. Thus instead of asking the question "What happens when this input sequence is applied", we can ask the question "What input sequence causes this system to fail".

References

[AB86] J. Archibald and J. L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *ACM/IEEE Design Automation Conference*, June 1990.

[BF89a] S. Bose and A. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *IEEE International Conference on Computer Design*, 1989.

[BF89b] Soumitra Bose and Allan L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In Luc Claesen, editor, *Proceedings of the IMEC-IFIP International*

Workshop on Applied Formal Methods For Correct VLSI Design, pages 759–764, November 1989.

[Bil87] J. P. Billon. Perfect normal forms for discrete functions. Technical Report 87019, BULL, March 1987.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[Bry88] Randal E. Bryant. Verifying a static ram design by logic simulation. In Jonathan Allen and F. Thomson Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 335–349. MIT Press, 1988.

[CBBG87] E. M. Clarke, S. Bose, M. C. Browne, and O. Grumberg. The design and verification of finite state hardware controllers. Technical Report 87-145, Carnegie Mellon University, 1987.

[CBM89] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In *9th International Symposium on Hardware Description Languages and their Applications*, 1989.

[KM89] R. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *ACM Symposium on Principles of Distributed Computing*, Edmonton, Alberta, 1989.

[Kur86] R. P. Kurshan. Testing containment of ω-regular languages. Technical Report 1121-861010-33-TM, Bell Laboratories, 1986.