Proceedings of the 1996 IEEE
International Conference on Robotics and Automation
Minneapolis, Minnesota - April 1996

Deadlock Prevention in Flexible Manufacturing Systems Using Symbolic Model Checking[1]

V. Hartonas-Garmhausen

Engineering & Public Policy Department
Carnegie Mellon University
Pittsburgh, PA 15213

E. M. Clarke, Sergio Campos

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper illustrates the use of symbolic model checking in the design of deadlock-free flexible manufacturing systems. Our verification methodology consists of the following stages. First, we extract a state machine model of the system. Second, we write the system specifications using a propositional temporal logic. Finally we use the model checker to check the state machine model of the system against its requirements. When a deadlock is identified, a counterexample is automatically generated with a scenario that leads to the deadlock. The counterexample is used to design the proper operational policy that will prevent the corresponding deadlock. This verification approach allows an exhaustive search of all possible behaviors and scenarios. We designed a flexible manufacturing system capable of producing 3 types of parts with 4 machines and 3 robots. It took 8 seconds to find possible deadlocks assuming machine processing capacity of one part at a time and about 36 seconds when we increased the machine processing capacity to two parts at a time. The size of the state space was in the order of 10^{18} states.

1 Introduction

This paper introduces a new approach for the design of flexible manufacturing systems using a temporal logic model checking verification system [2, 5, 6]. The significance of this approach is that it enables us to analyze industrial systems of realistic complexity. Model checking determines whether a state transition graph model of the system being verified satisfies system specifications expressed as formulas of a propositional temporal logic. In those cases where the specification is false, the model checker produces a counterexample execution path that shows why it is false. The counterexample is very useful in locating and correcting errors in the system.

Symbolic model checking is an industrial-strength formal specification and verification technique. It employs very efficient algorithms that allow an implicit representation of the transition relation using binary decision diagrams. Eliminating redundancy from the model and the need for an explicit enumeration of the states makes it possible to handle industrial applications. Industrial systems with up to 10^{30} states have been verified within minutes.

Flexible Manufacturing Systems (FMSs) employ multiple reprogrammable computer numerically controlled (CNC) machining centers, turning centers, and robots. A well-designed FMS combines high flexibility, maximum machine utilization, minimum in-process inventory, and maximum throughput. The design and analysis of FMSs is difficult because what makes these systems flexible also makes them complex. For example deadlocks, which are a common problem in FMSs, are often difficult to predict because they usually occur after a complex sequence of operations and inputs to the system. It is critical to avoid deadlocks since they lead to degraded performance bringing the entire system to a halt. Clearing deadlocks often requires human intervention which leads to higher labor costs.

---

[1] This research is sponsored in part by the National Science Foundation under grant no. CCR-8722633, by the Semiconductor Research Corporation under contract 92-DJ-294, by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under Grant F33615-93-1-1330.

Common approaches to deadlock detection include Petri nets [3, 4, 9], scheduling techniques [8], and a graph-theoretic procedure [10]. Deadlock prevention of FMSs using Petri nets involves an exhaustive path analysis of the reachability graph of the Petri net model of the system. Viswanadham et al. [4] concluded that deadlock prevention using Petri nets can only be implemented effectively for small systems. In real-world systems, the reachability graph may contain tens of thousands of states and arcs which makes the analysis of these graphs infeasible. By using ordered binary decision diagrams we are able to overcome this difficulty in most cases.

Model checking is a technique that can assist in identifying and eliminating deadlocks in large FMSs. Designing an FMS with the aid of model checking combines the two tasks of design and validation. Model checking can identify deadlocks and design errors very early in the design phase. By identifying errors in the early phase, it is possible to correct them at a significantly lower cost.

To demonstrate how are tools work, we verified a production cell consisting of 4 machines, 3 robots, and an L/U station that can produce 3 different part types. Using symbolic model checking we were able to identify several deadlocks and ways to prevent them.

The paper is organized as follows. Section 2 presents the formal logic and the model checking verification methodology. Section 3 demonstrates the use of symbolic model checking in the deadlock prevention of a flexible manufacturing system. Section 4 contains a number of conclusions.

2 Temporal Logic Model Checking

We model the FMS as a labeled state transition graph with each path corresponding to an execution trace of the actual system. This is possible because FMSs belong to the domain of Discrete Event Dynamic Systems (DEDS) in which the evolution of the system in time depends on the complex interactions of the timing of various discrete events, such as arrival of a part, departure of a finished part, or failure of a machine. The desired system specifications such as safety, reliability, operability, and performance are compiled from industrial quality standard checklists, field test checklists, process design specifications, hazard analysis, and other sources. The objective of temporal model checking is essentially to check whether a state transition system corresponding to a system description satisfies a desired specification expressed in temporal logic.

Temporal logic is a logic for expressing the ordering of events in time without introducing time explicitly. It contains operators for specifying properties such as "condition p will eventually hold" and "conditions p and q will never hold simultaneously." The particular temporal logic that we use is called CTL (Computation Tree Logic) [2]. Formulas in CTL are built from three components: atomic propositions, boolean connectives, and temporal operators. Atomic propositions refer to the values of the variables used to model the system. The boolean connectives are the standard ones, such as AND, OR, XOR, and NOT. Each temporal operator consists of two parts: a path quantifier (**A** or **E**) and a temporal modality (**F**, **G**, or **X**). All of the operators are interpreted relative to an implicit "current state." There are in general many execution paths (sequences of state transitions) of the model starting at this current state. The path quantifier indicates whether the modality denotes a property that should be true of all of those possible paths (the universal path quantifier **A**) or whether the property need only hold on some path (the existential path quantifier **E**). The modalities describe the ordering of events in time along an execution path and have the following intuitive meanings:

1. **F** ϕ (ϕ "holds sometime in the future") is true of a path if there exists a state on the path for which a formula ϕ is true.
2. **G** ϕ (ϕ "holds globally") means that ϕ is true at every state in the path.
3. **X** ϕ (ϕ "holds in the next state") means that ϕ is true in the state following the current state.

Figure 1 displays graphically some basic CTL expressions. A black circle indicates that a specification ϕ is TRUE in the corresponding state. A white circle indicates the reverse.

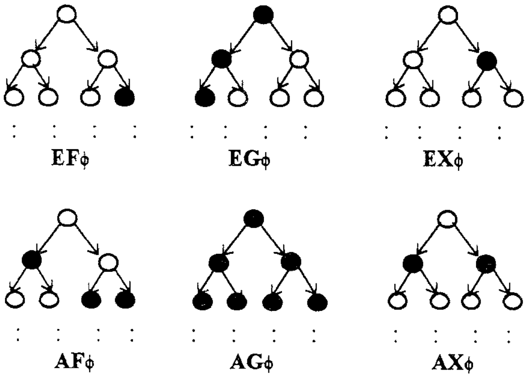

Figure 1 CTL Formulas

We represent states and transition relations using Ordered Binary Decision Diagrams (OBDDs) [1]. OBDDs generate very compact representations of formulas by eliminating redundancy and allowing computations on sets of states rather than on individual states. One way to understand OBDDs is to use the more familiar binary decision trees as reference. A binary decision tree is a rooted, directed tree that consists of terminal and nonterminal vertices. Each nonterminal vertex is labeled by a variable v and has two successors: *low(v)* if v is assigned *0* and *high(v)* if v is assigned *1*. Each terminal vertex is labeled by the value of v which is either *0* or *1*. Binary decision trees grow in size exponentially with the number of inputs and so they are only suitable for small state systems. OBDDs, which are also canonical forms for boolean equations, are similar to binary decision trees except that their directed acyclic graph structure allows nodes and substructures to be shared. We reduce binary decision trees into OBDDs by eliminating duplicate nodes and redundant tests. Figure 2 shows a simple binary decision tree and the corresponding more compact OBDD. Our model checking tool builds the OBDDs directly from the boolean equations automatically.

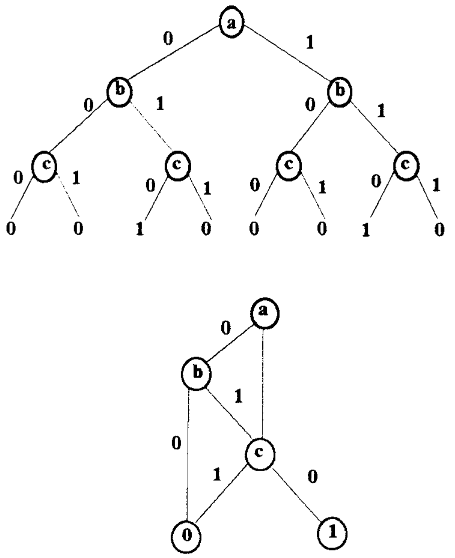

Figure 2 Constructing a Binary Decision Diagram for Binary Function $\bar{a}b\bar{c}+ac$.

As in the case of the binary decision tree, we traverse the OBDD starting at the top. At every nonterminal vertex, we follow the left or the right edge depending on the assignment (0 or 1) of the corresponding variable. The value encountered at the terminal vertex determines whether the corresponding truth assignment satisfies the formula.

The transition relation for the state transition graph is represented by an OBDD. The OBDD variables consist of two copies of the state variables, one for the current state and the other for the next state. To see if there is a transition from s to s', we simply assign the values of the state variables in state s to v and similarly for s' to v'. If the path in the OBDD for this assignment ends in the node labeled 1, then there is a transition from s to s'; otherwise there is no transition.

A state consists of a valuation for the variables, plus a valuation for the inputs. Each formula of the logic is either true or false in a given state. An atomic proposition corresponding to a variable x is true in a state if x has the value 1 in that state and false if it has the value 0. A formula is built from atomic propositions using boolean connectives. Often it is more intuitive to think in terms of sets of states instead of formulas. When we consider a formula we are interested in the set of states that satisfy it. We can find these states easily representing the formula by an OBDD. A state satisfies the conjunction of two formulas if and only if it is in the intersection of the sets identified by the formulas. The same principle applies to disjunction, union, complement, and negation.

The size of an OBDD depends critically on the variable ordering, i.e. the order of the variables as they are encountered in the traversal of the OBDD from root to nodes. Several heuristics have been developed for finding a good variable ordering while finding the optimal ordering is in general NP-complete. The intuition of these heuristics comes from the observation that OBDDs tend to be small when related variables are close together in the ordering. When no obvious ordering heuristics apply, we use a technique called dynamic reordering which reorders the variables periodically to reduce the number of vertices in use.

3 Flexible Manufacturing System

We use the following production cell to illustrate our method. There are 4 numerically controlled machines M1, M2, M3, and M4. Each machine can process two parts at a time. This FMS processes parts of three types P1, P2, and P3. The processing route of a part depends on its type. There may be several alternative routes a workpiece of a given type can traverse which adds flexibility to the system. There are three robots R1, R2, and R3 with the capacity to transport and shuttle parts to the appropriate machines, wait for machines to become available, and finally unload finished parts to the L/U station. Each robot has its own action area. R1 loads parts of type P1, serves machines M1, M2, and M4, and unloads finished parts of type P2. R2 loads raw parts of type P2, serves machines M1, M2, M3, and M4, and unloads finished parts of type P3. Finally, R3 loads raw parts of type P3, serves machines M3 and M4 and unloads finished parts of type P1. The main controller guides the operation of the system given the availability of robots, machines, and raw parts at the Load/Unload (L/U) station. Its role is to manage the system by scheduling, dispatching, monitoring the system resources, and reacting to machine breakdowns, broken tools, and deadlocks.

We use symbolic model checking to design the FMS and determine the necessary operational policies such that deadlocks never occur. Using this method we can also determine early in the design phase the impact different policies have on the performance of the system. We can compare alternative routes within the system or different policies for when to introduce a part type into the system. For a more detailed analysis of the system performance subsequent models will also include detailed processing and transport times, rework of parts, and machine failures.

We start by describing the behavior of the system with boolean equations. We can describe synchronous or asynchronous systems, detailed deterministic or abstract nondeterministic models. The SMV language supports finite data types (booleans, scalars, and fixed arrays) and uses expressions in the propositional calculus to describe the transition relation of the finite state machine. Figure 3 lists a part of the FMS model.

```
MODULE Machine_I
VAR
cmd : {process_P1, process_P3, release_P1,
      release_P3, rest};
state : {Machine_available, Machine_processes_P1,
      Machine_processes_P3,
      Machine_waits_for_R2_with_P1,
      Machine_waits_for_R2_with_P3};
ASSIGN
init(state) := Machine_available;
next(state) :=
  case
  cmd=process_P1 : Machine_processes_P1;
  cmd=process_P3 : Machine_processes_P3;
  cmd=release_P1 :
              Machine_waits_for_R2_with_P1;
  cmd=release_P3 :
              Machine_waits_for_R2_with_P3;
  cmd=rest : Machine available;
  1 : state;
  esac;
```

Figure 3 Machine Module

A *MODULE* is an encapsulated group of declarations. Modules can be reused and can also be parameterized so that each instance of a module can use different data values. The *VAR* declaration creates an instance of a module and variables. A module can contain instances of other modules and thus we can model the structural hierarchy of a system architecture. In the main FMS controller module we built the processing slots of Machines M1 and M2 by instantiating the Machine_I four times.

The state of the model is defined by a set of state variables that may be of boolean or scalar type. The variable *cmd* in *MODULE Machine_I* is declared to be a scalar that can take on the symbolic values *process_P1*, *process_P3*, *release_P1*, *release_P3*, and *rest*. The value of a scalar variable is encoded by the interpreter using boolean variables so that the transition relation may be represented by an OBDD.

The keyword *ASSIGN* introduces the parallel assignments that determine the initial states and the transition relation. In the *Machine_I* module, the initial value of the variable *state* is *Machine_available* and the next value of the machine's state is determined by the current state of the system and the assigned value of the case expression. The variable is assigned the value on the right hand side when the corresponding expression on the left hand side is true. A variable can also be assigned a set of values in which case the result is a non-deterministic choice among the elements of the set. Non-deterministic assignments are useful for modeling abstract models of complex systems and for modeling systems that are not yet fully implemented. Figure 4 shows that the controller may choose to deliver to either machine M1 or machine M2 when robot carries a part of type P1 and both machines M1 and M2 are available.

```
next(R1.cmd) := case
            next(R1.state) = Robot_carries_P1 &
            next(M1_available) &
            next(M2_available):
            {deliver_to_M1, deliver_to_M2};
            ...
```

Figure 4 Non-deterministic variable assignment

The model checker constructs the OBDDs corresponding to the boolean equations. By default, all of the assignments are executed simultaneously. There are also techniques available for defining parallel processes that are interleaved in the program execution, which is useful in modeling systems where actions are not synchronized.

Next we check properties of the system. For deadlock prevention we need to identify the necessary conditions that lead to deadlocks and then design operational policies such that deadlocks will never occur.

We check whether any of the following conditions arises:

- A resource that is used by two or more processes simultaneously.

- A resource that is released before the process using it is complete.
- A process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.
- A set of waiting processes $\{p_1, \ldots p_m\}$ such that p_i is waiting for a resource held by process p_{i+1} for i=1, m-1 and p_m is waiting for a resource held by process by p1.

We express the above conditions as CTL formulas. For example, the specification *AG !(R2.state= waits_for_M3 with_rawP2 & M3.state= waits_forR2_with_finishedP2)* states that it is never the case that robot R2 will wait for machine M3 to shuttle a new raw part P2 while M3 waits for R2 to unload a finished part P2. This is a circular wait condition.

The model checker found several possible deadlocks and generated traces to demonstrate how the system reaches a deadlock in each case. These scenarios helped us debug the FMS controller and design resource allocation policies that ensure deadlock prevention. Figure 5 describes one of the deadlocks that we identified. We have included those variables that are most relevant to the deadlock.

```
--specification AG R1.cmd=unload_P2 is false
--as demonstrated by the following execution sequence

    state 1.1
    R1.cmd = load_raw_P1
    R2.cmd = load_raw_P2
    R3.cmd = load_raw_P3
    M1_a.state = Machine_available
    M2_a.state = Machine_available
    M3_a.state = Machine_available
    M4_a.state = Machine_available

    state 1.2
    R1.cmd = deliver_P1_to_M1_a
    R2.cmd = deliver_P2_to_M3_a
    R3.cmd = deliver_P3_to_M4_a

    state 1.3
    R1.cmd = load_raw_P1
    R2.cmd = load_raw_P2
    R3.cmd = load_raw_P3
    M1_a.state = Machine_processes_P1
    M3_a.state = Machine_processes_P2
    M4_a.state = Machine_processes_P3

    state 1.4
    R1.cmd = deliver_P1_to_M2_a
    R2.cmd = deliver_P2_to_M3_b
    R3.cmd = deliver_P3_to_M4_b
    M1_a.state = Machine_waits_for_R2_with_P1
    M3_a.state = Machine_waits_for_R2
    M4_a.state = Machine_waits_for_R1_with_P3

    state 1.5
    R1.cmd = load_P3_from_M4_a
    R2.cmd = load_P1_from_M1_a
    M2_a.state = Machine_processes_P1
    M3_b.state = Machine_processes_P2
    M4_b.state = Machine_processes_P3

    state 1.6
    R1.cmd = deliver_P3_to_M1_a
    R2.cmd = wait_for_M3
    R3.cmd = load_raw_P3
    M2_a.state = Machine_waits_for_R2_with_P1
    M3_b .state= Machine_waits_for_R2
    M4_b.state = Machine_waits_for_R1_with_P3

    state 1.7
    R1.cmd = load_P3_from_M4_b
    R3.cmd = deliver_P3_to_M4_a
    M1_a.state = Machine_processes_P3

    state 1.8
    R1.cmd = deliver_P3_to_M1_b
    R3.cmd = load_raw_P3
    M1_a.state = Machine_waits_for_R2_with_P3
    M4_a.state = Machine_processes_P3
```

Figure 5 A deadlock

Machine M1 waits for robot R2 to unload the workpieces (types P1 and P3) it finished processing. Robot R2 is waiting for machine M3 to become available so that it can deliver to M3 a workpiece of type P1. However, M3 requires R2 to release the parts it finished processing. The FMS is deadlocked because M1, R2, and M3 can not release their resources and will wait indefinitely for each other to become available.

To prevent this deadlock we added the following policies to the control of robot R2.

1. When machines M3 and M1 (or M2) wait for robot R2 simultaneously, R2 will first be dispatched to serve machine M3.
2. While M3 is busy, R2 will not unload machines M1 and M2 of workpieces of type P1.

This type of analysis helps us understand better the behavior of the system. Understanding how a deadlock comes about helps us make the right design changes to prevent it.

4 Conclusions

Symbolic model checking is a powerful formal specification and verification method that has been applied successfully in the verification of several industrial designs [7, 8]. This paper describes how this method can be used to design and formally verify flexible manufacturing systems.

The state explosion problem which arises from the complexity of flexible manufacturing systems makes the analysis of these systems a difficult task. By representing them implicitly using ordered binary decision diagrams, symbolic model checking can handle in many cases the state explosion problem.

The model discussed above consisting of an L/U station, 4 machines, and 3 robots took about 8 seconds to be verified assuming that each machine can process one part at a time. Changing the model to increase the machine processing capacity to two parts at a time lead to a more complex model with a state space in the order of 10^{18} states. After finding a good variable ordering, which critically reduces the size of the OBDD, the model checker identified a deadlock within 36 seconds. Several deadlocks were detected. The counterexamples that were generated helped us design the appropriate policies to avoid these deadlocks.

Symbolic model checking has several advantages.
It allows an exhaustive search of all possible behaviors of the system in an automatic fashion. It provides quick feedback in the evaluation of design alternatives. It can find design errors that may be missed by simulation techniques if they occur after a complex sequence of input changes. The symbolic model checking algorithms allow the verification of realistic complex applications in an efficient manner which makes them a great tool in the design and verification of flexible manufacturing systems.

References

1. E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Trans. Comp. C35(8), 1986.
2. 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.
3. N. Viswanadham, Y. Narahari, T. L. Johnson. Deadlock Prevention and Deadlock Avoidance in Flexible Manufacturing Systems Using Petri Net Models. IEEE Transactions on Robotics and Automation, Vol. 6, No. 6, pg. 713-723, Dec. 1990.
4. N. Viswanadham, Y. Narahari. Performance Modeling of Automated Manufacturing Systems. PRENTICE HALL, 1992.
5. K. L. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. Ph.D. Thesis, Carnegie Mellon University, 1992.
6. E. M. Clarke, O. Grumberg, D. Long. Verification Tools for Finite-State Concurrent Systems. Proceedings of A Decade of Concurrency: Reflections and Perspectives REX School/ Symposium, Noordwijkerhout, The Netherlands, 1-4 June 1993.
7. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In Proceedings of the 11th CHDL, 1993.
8. V. Hartonas-Garmhausen, T. Kurfess, E. M. Clarke, D. Long. Automatic Verification of Industrial Designs. In Proceedings of the Workshop on Industrial Strength Formal Specification Techniques, pg. 88-96, April 1995.
9. S. Ramaswamy, S.B. Joshi. Deadlock Avoidance in Automated Manufacturing Workstations - A Scheduling Approach. Proceedings of 1994 IEEE International Conference on Robotics and Automation, pg. 1992-1997, 1994.
10. J. Espeleta, J.M. Colom, J. Martinez. A Petri Net Based Deadlock Prevention Policy for Flexible Manufacturing Systems. IEEE Transactions on Robotics and Automation, Vol. 11, No. 2, pg. 173-184, April 1995.
11. H. Cho, T. K. Kumaran, R. Wysk. Graph Theoretic Deadlock Detection and Resolution for Flexible Manufacturing Systems. IEEE Transactions on Robotics and Automation, Vol. 11, No. 3, pg. 413-421, June 1995.