Experimental Results

When all was said and done, the new algorithm was run using both the Lazy and Strict policies using the SMART tool and compared to the results of previous MDD-based algorithms also run in SMART. For the models that the algorithms were ran on, this papers' approach is up to two orders of magnitude faster than the Recursive algorithm, and up to one order of magnitude faster than the forwarding algorithm.

The Saturation algorithm's required peak memory is often close to the final memory needed for storing the overall state space.

Reguarding Generation time, in order from best to worst: 1) Lazy 2) Strict(100) 3) Strict(1) 4) Forwarding 5) Recursive

Reguarding Memory Consumption 1) Strict(1) 2) Strict(100) 3) Lazy 4) Forwarding 5) Recursive

- The threshold that triggers recycling can be set in terms of number of nodes or bytes of memory. The policy using a threshold of one node, denoted as STRICT(1), is optimal in terms of memory consumption, but has a heigher overhead due to more frequent clean-ups.

STRICT Policy

- Disconnected nodes has a delete-flag set and its arcs k.p[i] are re-direct to (k - 1.0) with possible recursive effects on the nodes downstream.

- When a hit in the union cache or firing cache returns s, we consider this entry stale if the delete flag is set.

- By keeping per-level count of nodes with delete-flags set, the decision can be made whether to a) allocate new memory for a node at level k or b) recycle the indices and physical memory of all nodes at level k with delete-flag set after removing all the entries in Union Cache and Firing Cache referring to them.

Garbage Collection

- MDD nodes can become disconnect, unreachable from root
- Detected by using an incoming-arc counter for each node
- LAZY policy - gives best runtime by removing these disconnected nodes only at the end.

Given MDD encoding of the initial state s, we saturate its nodes bottom-up. This improves both memory and execution-time efficiency for generating state spaces because of the following reasons:

1) Saturation order ensures that the firing of an event affecting only the current and possibly lower levels adds as many new states as possible

2) Since each node in the final encoding of S is saturated, any node inserted in the unique table has a chance of being part of the final MDD.

3) Once we saturate a node at level k, there is no need to fire any event e in it again

Key Ideas

1) Fire events node-wise and exhaustively, instead of level-wise and just once per iteration

2) Use a unique table to detect duplicate nodes

3) Use operation caches such as a union cache and a firing cache to speed up computation

4) Only saturated nodes are checked in the unique table or referenced in the caches

Algorithm Employing Node Saturation

In previous work, a naive strategy was used that cycled through MDDs level-by-level and fired, at each level k, all events e with First(e) = k.

Saturation not only simplifies this algorithm, but also significantly improves its time and space efficiency

- Level 0 consists of two terminal nodes $<$0.**0**$>$ and $<$0.**1**$>$

- A non-terminal node $<$k.p$>$ has n^k arcs pointing to nodes at level k - 1

- A non-terminal node cannot duplicate another node at the same level

- Given a node $<$k.p$>$, we can recursively define the node reached from it through any integer sequence using $Beta$ and $Alpha$ the nodes below and above $<$k.p$>$ respectively.

Objective Want to use efficient data structures to encode S that exploit the system's structure

Solution Multi-valued decision diagrams!

Notation Note: Superscripts are used for submodel indices, not for exponentiation. Subscripts are used for event indices.

- Sytem model is composed of K submodels
- Nodes are organized into K + 1 levels
- Level K contains only a single non-terminal node $\langle k.r \rangle$, the root
- Levels K - 1 through 1 contain one or more non-terminal nodes

State Spaces and Next-State functions

A discrete-state model must specify i) S hat, the potential states ii) s , initial state that must belong in S hat iii) N, the next state function describing what states can be reached from a given state in a single step

S is the reachable state space. In this paper we assume that S is finite; however, for most practical asynchronous systems, the size of S is enormous due to the state-space explosion problem.

- In most concurrency frameworks, next-state functions satisfly a product form that allow each component of the state vector to be updated somewhat independently showing significant improvements in speed and memory consumption when compared to other state-space generators.

New Knowledge:

- The reachable state space of a system can be built by firing the system's events in any order, as long as every event is considered often enough.

- This paper introduces a strategy which exhaustively fires all events affecting a given MDD node, thereby bringing it to its final "saturated" shape.

- Compared to previous work, saturation eliminates amount of administration overhead, reduces the average number of firing events, and enables a simpler more efficient cache management.

Problem

Asynchronous systems, such as communication protocols, suffer from state-space explosion.

Solution

Problem has been addressed in previous work by using Multi-valued Decision Diagrams (MDDs)

MDDs in event-based asynchronous systems

Knowledge in Previous Work:

- Each event updates just a few components of a system's vector.

- Firing an event only requires the application of local next-state functions and the local manipulation of MDDs.

- BDDs construct their state spaces by iteratively applying a single, global next-state function which is itself encoded as a BDD.

Introduction

- State-space generation used in formal verification tools

- Today's high complexity of digital systems require huge state spaces in the small memory of a work-station

- Decision Diagrams are a kind of data structure for implicitly representing large sets of states in a compact fashion and therefore a rather logical choice for data structure.

- Binary Decision Diagrams (BDD) have proven very successful for synchronous systems, such as digital circuits, increasing the manageable size from 10^6 states to 10^{20} states.

Purpose

- Present algorithm for generating state spaces of asynchronous systems using Multi-valued Decision Diagrams (MDD)

- Accomplished through a new strategy called saturation and implemented in the tool SMART

Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation

Gianfranco Ciardo, Gerald Luettgen,
and Radu Siminiceanu.
Departments of Computer Science
College of William and Mary, and Sheffield University

Kerry Connell
College of William and Mary

April 2 2002