Efficient State-exploration

J. Geldenhuys
Department of Computer Science
University of Stellenbosch, Stellenbosch 7600

Abstract

The exploration of states is an important element of many problems. Many problems share properties which allow the formulation of general strategies for state exploration. This article examines these strategies with reference to the problem of deadlock detection in labelled transition systems. The main issues are discussed and four major tasks are identified: state generation, scheduling, state storage, and state compaction. Empirical data is presented, optimizations and special restrictions are discussed and, finally, the results are generalized.

1 Introduction

The exploration of states is an important element of many problems. In many cases the efficiency of a solution depends on the state exploration techniques employed. A significant number of these problems share properties which can be exploited to select more efficient search techniques. Examples of such problems are graph algorithms, model checking, and optimization problems. This article examines these strategies with reference to the specific problem of detecting deadlock in a labelled transition system.

Important issues such as depth-first vs. breath-first exploration, on-the-fly techniques, the effect of interleavings and optimizations are investigated. Secondly, four major tasks are identified: *state generation*, *scheduling*, *state storage*, and *state compaction*. Thirdly, empirical data is presented to illustrate the issues raised and their application to deadlock detection. Possible optimizations (e.g., using partial order methods to prune the state graph), and special restrictions (e.g., fairness) are discussed and, finally, the results are generalized.

2 The problem

A state is a canonical description of the status of a system. It uniquely identifies the values of program counters, variables, queue contents and other data structures. To solution to many problems is the identification of a single state which has a certain property. For example, in computer chess the object is to find the state in which the potential for a win is at a maximum. A significant number of these problems share the following properties:

- The state space is extremely large, states themselves are extremely large, and/or states are time consuming to generate.
- The entire state graph may have to be traversed to solve the problem.
- It is possible to ignore subgraphs based on information obtained during runtime.
- When a state is revisited, the revisited subgraph is not re-explored.
- A non-probabilistic solution is preferred/required.

Examples of such problems are graph algorithms (e.g., Tarjan's algorithm for finding strongly connected components), model checking, optimization problems (e.g., network routing, computer chess, and scheduling), theorem proving, and code optimization. The rest of this article will focus on the deadlock detection problem for labelled transition systems as a typical example.

Deadlock detection

A labelled transition system (LTS) is a tuple (S, Σ, Δ, s_0) where S is a set of *states*, Σ is a set of *actions* (or *transitions*), the *next state relation* $\Delta \subseteq S \times \Sigma \times S$, and the *initial state* $s_0 \in S$. An LTS can be represented as a state graph by taking S as nodes and Σ as edges. A *path* is a sequence of states $s_1, s_2, s_3, \ldots$ so that for each i there is a transition $t_i \in \Sigma$ with $(s_i, t_i, s_{i+1}) \in \Delta$. Paths may be infinite. To detect deadlock in an LTS the state graph is searched for a node with an out-degree of 0, or equivalently, with no successors. Such a node represents a state with no enabled transitions, i.e., a deadlock state. We will restrict ourselves to LTSs with a finite set of states, but techniques for infinite LTSs exist [4]. An LTS provides a convenient way to describe programs and to check their properties.

Important issues

How are deadlock states found? One possibility is simulating the behaviour of the LTS. The algorithm starts in state s_0 and randomly selects transitions to execute. The resulting path of states is a random walk through the state graph which terminates when a deadlock state is found. Heuristics can be used to guide the selection of transitions in order to improve the probability of finding deadlock. However, the process may carry on for an indefinite time. It cannot be shown that an LTS is deadlock-free unless all its states have been visited. If all states must be visited, they may as well be generated systematically.

In [5] the systematic generation of states is accomplished by calculating the state graph, storing it in memory, and then checking its properties. Deadlock may occur after the first few transitions, but unfortunately the entire state graph must be calculated for every run. It also places an upper bound on the size of LTSs that can be checked since the entire state graph must fit into memory.

It is a better idea to make the search dynamic: states must be generated on the fly as the state graph is explored. Cycles in the state graph must be detected. If no provision is made for cycle detection, the search will not terminate but will re-explore the first cycle it finds continuously. There are two methods of exploration: breadth-first and depth-first. Breadth-first search is useful for finding the shortest path that violates the specification. This is desirable but not essential in detecting deadlock. Unfortunately, breadth-first not only requires more space than depth-first search, but cycles in the state graph can only be detected by comparing each new state to all previously visited states. States can also be represented implicitly, e.g., through equivalences classes. Impressive results have been reported for symbolic model checking, a technique based on breadth-first search and implicit representation [3]. Depth-first search is the method of choice for deadlock detection and model checking algorithms. A single path is maintained on a stack. New states are pushed onto the stack as they are explored and popped from the stack once they have been fully explored.

The central problem associated with searching problem spaces is that of *state explosion*: the number of states grows exponentially in the number of processes and the degree of non-determinism. The cause of this is that all interleavings of transitions must be explored. A set of n independent transitions can be ordered in any of $n!$ ways. One approach to this problem is to break an LTS up into smaller subsystems and to check them independently. An important group of reduction techniques is based on equivalence between the original and reduced state graphs [12]. The equivalence makes it possible to derive a set of conditions under which certain paths can be ignored.

3 Tasks

State generation

To describe an LTS a modeling language ESML (Extended State Machine Language) is used. ESML is based on CSP [8] and Joyce [1] and was designed to facilitate complex data structures. Subranges, enumeration types, records, array and lists are supported. An example of a small model is shown in Figure 1.

In a first implementation the ESML code was translated to Modula-2 and compiled to form the state generation module. This module is then linked to the rest of the system and executed to detect deadlock. This technique is standard practice [10, 13]. However, the translation of code from one level of abstraction to another is complex and therefore it is difficult to find errors.

```
MODEL ProducerConsumer;

TYPE int = 0..1; chan = {msg(int)};
VAR ch: chan;

PROCESS Producer(OUT c: chan);
BEGIN
  DO TRUE -> c!msg(0)
  [] TRUE -> c!msg(1)
  END
END Producer;

PROCESS Consumer(IN c: chan);
VAR x: int;
BEGIN
  DO TRUE -> c?msg(x) END
END Consumer;

BEGIN
  Producer(ch); Consumer(ch)
END ProducerConsumer.
```

Figure 1: *An ESML model of a producer and consumer.*

More recently the use of an abstract machine was investigated [6]. An abstract machine was designed to support model checking. ESML is translated to abstract machine instructions that are executed by an interpreter to generate states. This approach has proven very successful in the domain of compilers [11, 14]. The code for this system is much simpler and more reliable since instructions can be shown to be correct independently of one another. The abstract machine has 47 instructions and supports multiple processes, inter-process communication, complex data structures such as lists and non-deterministic choice. It has a memory of 4000 words which is used to store the program code, variables and stack for evaluating expressions. The variables are compacted to form the *state* which corresponds to the state of the LTS (Figure 2).

The machine performs two basic operations: it can *Execute* to produce the next state from the current state. If the next state has already been explored (and is being revisited), or forms a cycle in the graph, or does not exist, the machine falls back into the previous state and returns a value to indicate what it is doing. It can also *Backtrack* into the previous state.

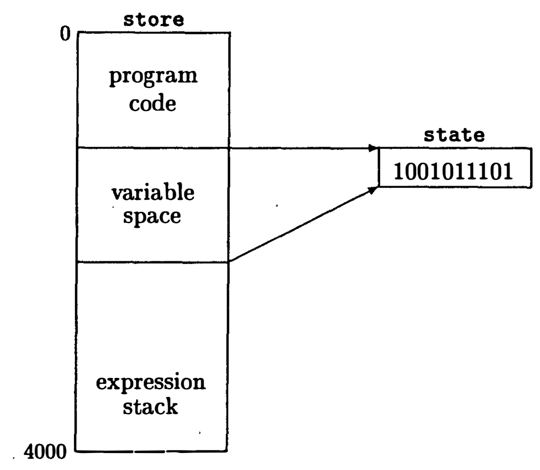

Figure 2: *The abstract machine memory layout.*

Scheduling

As the depth-first search algorithm explores the state graph, the states are stored on a stack. As a new state is explored, it is pushed onto the stack. Once a state has been fully explored it is removed from the stack. The stack contains the path that is currently being explored. If deadlock is detected, the stack is dumped to provide the user with the path that led to the error. Storage of the current path allows the system to detect cycles. When a new state is added to the stack, it is first checked to make sure that is not already present. If it is present, a cycle is reported and the state is not added to the stack.

The current path plays an important role when falling back into states. Once a state has been fully explored, the machine state must be restored to that of the predecessor state so that other states can be explored. The state itself is already stored on the trace, but the machine has other data structures that must also be returned to their prior state. The program code does not change and the expression stack is always empty when a transition executes. To store the entire variable space would be impossible since it can be very large. It is possible to extract the variable space from the compacted state, but this would be time-consuming. The last option is to store only the changes made to the variable store. As variables are changed, the compacted state is updated and the changes are recorded on the stack. When falling back, the necessary changes are made undone.

If the machine is returned to exactly its previous state, it would select the

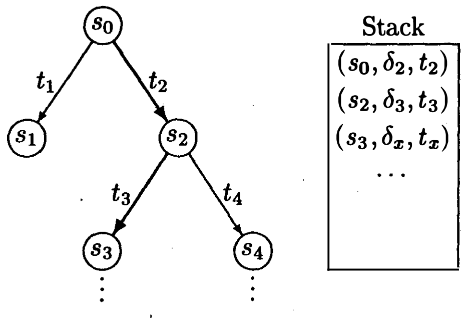

Figure 3: *The stack contains the current path*

same transition over and over again. The stack therefore also stores the last transition executed. When falling back this information is used to select the correct next transition. In this way transitions are scheduled by the machine.

State storage

As explained above, the depth-first algorithm will re-explore a state if it is revisited. To prevent this unnecessary work, visited states must be stored. Several options exist:

One possibility is the use of binary decision diagrams (BDDs) [2]. A BDD is a data structure that represents a boolean function, or alternatively a set of binary words. Each time a state is visited, its compacted state is added to the BDD. An insert operation takes $O(m)$ time, and a lookup operation $O(\log m)$, where m is the size of the diagram. The size depends on the contents of the BDD: if entries are similar, the diagram is probably compact; if entries differ greatly, it may grow to up to 2^{n-1} nodes (n is the length of the state vector in bits). When the diagram grows too large to fit into memory, the search has to be abandoned, because states cannot be deleted. As the size grows, so does lookup and insert time. Unfortunately it is difficult to predict the behaviour of a BDD. The order of the variables play an important role, but to find the optimal order is prohibitively expensive. BDDs are also known to perform badly when they are used to store counter variables. Therefore BDDs are not suitable for the storage of states.

A second possibility is the bit vector technique [9]. It uses the available memory as a large array of bits. A state is hashed to an address (or *bit vector*) into the array and the particular bit is set. When the state is encountered again, the bit is tested to detect whether the state is being revisited. In this way each state is associated with a single bit. The technique relies on the distribution of the hash function to prevent collisions. If, however, two states hash to the same address, the first state will set the bit when it is visited, and the second state will be erroneously ignored. Although the probability of a collision is small, it grows as the number of states grow. Of course, the user will never know if a collision has occurred and must always recheck results. Suggestions have been made on how to improve the reliability of this method (using two bits per state, for example) [15], but it remains an approximate technique and is usually only resorted to when other techniques fail.

The third and final possibility is a cache of states. Unlike the first two methods, a cache stores states explicitly in a table. It therefore requires more memory per state. It is almost as fast as the bit vector technique: hashing is used to locate entries in a table of states. Collisions are resolved through open hashing: states are rehashed until they are found in the cache or an open slot is found in which to insert them. As the cache grows fuller, collisions increase and it becomes necessary to replace older states to insert new ones. A random replacement strategy was found to be the most effective. When states are replaced and subsequently revisited, they are not found and are re-explored. In such cases unnecessary work cannot be avoided but the re-exploration of the subgraph does not continue long since successor states will still be present in the cache. It has been shown that a cache works well for models with 2–3 times the number of states in the cache[7].

As a further optimization we have incorporated the stack in the cache. This means that new states are not searched for twice (in the stack and in the cache) but once only. A skeleton stack is used to maintain the order of the stack entries, but the states are stored along with the other cached states. Stack states are never overwritten because when the depth-first search falls back into them, they have to be present.

State compaction

Since the size of memory is such a serious limitation of the technique, it makes sense to expend effort to make states as small as possible. Compression algorithms (e.g., Huffman encoding, the Lempel-Ziv family of algorithms) can reduce state vector size, but are unsuitable for two reasons: (1) compressed states would vary in length and would consequently be difficult to manipulate, compare, and store efficiently; and (2) compression would slow the system down, since after every transition the entire state has to be compressed, even when only a small part of it has changed.

An alternative method, called state compaction, attempts to minimize the number of bits assigned per variable. few bits per variable as possible. Consider the following declarations:

```
VAR
   c: (red, green, blue, white, black);
   b: BOOLEAN;
```

```
p: RECORD x, y: 0..149 END;
```

Typically a compiler for a programming language, allocates storage space in multiples of 8 bits (1 byte) and would allocate $4 \times 8 = 48$ bits for the declarations above. At this level bits are wasted, e.g., variable c is allocated 8 bits but uses only 5 values out of a possible 256. The deadlock detection system needs to be more frugal when it comes to memory allocation, and must allocate variable storage space in multiples of 1 bit. When bits are assigned per variable, a variable of size s can fit into $\lceil \log_2 s \rceil$ bits. This method would allocate 20 bits to the variables above.

Storage allocation can be optimized even further. Record p has been allocated 16 bits in all three cases above. Its fields x and y, however, can only assume 150 values each, and 150×150 unique values will fit into $\lceil \log_2 150^2 \rceil = 15$ bits. When this idea is applied to the declarations as a whole, they can be stored in $\lceil \log_2 (5 \times 2 \times 150 \times 150) \rceil = 18$ bits.

A compacted state containing the variables above is calculated as

$$\begin{aligned} S &= \mathtt{c} + 5 \cdot (\mathtt{b} + 2 \cdot (\mathtt{p.x} + 150 \cdot \mathtt{p.y})) \\ &= \mathtt{c} + 5 \cdot \mathtt{b} + 5 \cdot 2 \cdot \mathtt{p.x} + 5 \cdot 2 \cdot 150 \cdot \mathtt{p.y} \end{aligned}$$

It is clear from the first line that c can range over its five values 0...4 without affecting the other variables. Similarly, the other variables can range over their respective values without influencing c. Variable b for instance, can assume its values 0 and 1 without affecting c or the variables to the right. Associated with each variable z are a lower factor $\mathtt{z}_l$ and higher factor $\mathtt{z}_h$, e.g., for variable b, $\mathtt{b}_l = 5$ and $\mathtt{b}_h = 5 \cdot 2 = 10$. The lower factor "shields" the smaller terms to the left and is useful for isolating these terms, e.g., $S \bmod \mathtt{b}_l = \mathtt{c}$. In the same way the higher factor can be used to obtain a certain term, e.g., $S \bmod \mathtt{b}_h = \mathtt{c} + 5 \cdot \mathtt{b}$ from which b is obtained by dividing by $\mathtt{b}_l$: $(\mathtt{c} + 5 \cdot \mathtt{b}) \operatorname{div} \mathtt{b}_l = \mathtt{b}$. Two basic operations must be performed on S:

1. To obtain the value of variable z the higher factor is used to strip out all variables to the right of z and the lower factor is used to strip out all variables to the left of z:

$$\mathtt{z} = (S \bmod \mathtt{z}_h) \operatorname{div} \mathtt{z}_l$$

2. To change the value of a variable, if the value of z changes to z′, the updated state vector is

$$S' = S + \mathtt{z}_l \cdot (\mathtt{z}' - \mathtt{z})$$

This operation is the combination of two steps: first the old value is removed (by subtracting $\mathtt{z}_l \cdot \mathtt{z}$) and then the new value is inserted (by adding $\mathtt{z}_l \cdot \mathtt{z}'$).

There is no run-time overhead involved in calculating the lower and higher factors: they are computed beforehand and stored in a table. The cost of a variable lookup is therefore two multiplications (operation 1) and that of a variable change is two additions and a multiplication (operation 2).

The idea of state compaction was first explained in [13] where its use was limited to complex data structures such as lists, arrays and records. In the current implementation, state compaction is applied to entire states with great effect. In the example above the state was compacted to 38% of the size of the original. In general this percentage varies from

4 Implementation

The system described above was implemented in Oberon. The Oberon language was chosen because it is highly portable and runs on a range of platforms.

Module	Lines	Code size
State generator	803	17132
Scheduler	221	3040
Storage	142	1568
Compaction	161	2196
Total	**1327**	**23936**

The system was used to check for deadlock in a model of the classical dining philosophers problem, a model of an elevator system, and a model of a process scheduler. The following results were obtained on a Silicon Graphics machine with 64Mbytes of memory and a 150MHz processor:

Model	States	Trans.	Time	Trans./sec
Dining philosophers	420096	2338593	123.52	18932.91
Elevator system	1633032	9086599	439.81	20660.28
Process scheduler	2016168	9908147	450.07	22014.68

The process scheduler model produced nearly 10^7 transitions and was checked at 22000 thousand transitions per second. Models of up to 4.2×10^6 states (19.4×10^6 transitions) have been checked to date on the same machine. These figures are on par with state of the art systems and represent the current limits of these techniques.

Analysis of the system during a typical run shows the following use of time:

Module	% of time
State generator	50.43
Scheduler	10.90
Storage	21.63
Compaction	15.17
Other	1.87

As expec   e generation of states takes the most time. For most of the other problems mentioned in Section 2 state generation is less expensive in terms of time and even more states can be explored per second.

5 Conclusion

The techniques that have been described are based on the properties outlined in Section 2. With a little modification they can be applied to a wide range of problems. State generation is the central issue when addressing a new problem. In the case of deadlock detection this was addressed throught the use of an abstract machine. The scheduling and storage mechanism is general enough to be reused in other cases, but state compaction is based on our knowledge of what a state looks like. It is however very important to perform some form of compaction on states, in order to obtain satisfactory results.

References

[1] P. Brinch Hansen. Joyce—A Programming Language for Distributed Systems. *Software—Practice and Experience*, 17(1):29–50, January 1987.

[2] R. E. Bryant. Graph-Based Algorithm for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8), August 1986.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[4] O. Burkart and Y.-M. Quemener. Model-Checking of Infinite Graphs Defined by Graph Grammars. Technical Report 995, Institut de Recherche en Informatique et Systèmes Aléatoires, May 1996.

[5] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Lecture Notes in Computer Science #131*. Springer-Verlag, 1981.

[6] J. Geldenhuys. A Reliable and Efficient Abstract Machine for Program Verification. Master's thesis, University of Stellenbosch, In preparation.

[7] P. Godefroid, G. J. Holzmann, and D. Pirottin. State Space Caching Revisited. In *CAV'92: Proceedings of the 4th International Conference on Computer-Aided Verification*, pages 175–186, June 1992.

[8] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[9] G. J. Holzmann. An Imporved Protocol Reachability Analysis Technique. *Software—Practice and Experience*, 18(2):137–161, February 1988.

[10] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software Series, 1991.

[11] A. S. Tanenbaum. Implications of Structured Programming for Machine Architecture. *Communications of the ACM*, 21(3):237–246, 1978.

[12] A. Valmari. Error Detection by Reduced Reachability Graph Generation. In *Proceedings of the 9th European Workshop on Application and Theory of Petri Nets*, pages 95–112, 1988.

[13] W. C. Visser. A Run-time Environment for a Validation Language. Master's thesis, University of Stellenbosch, October 1993.

[14] N. Wirth. Pascal-S: A Subset and Its Implementation. Technical report, ETH, Zürich, June 1975.

[15] P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *CAV'93: Proceedings of the 5th International Conference on Computer-Aided Verification*, 1993.