custbld.slm

```
import builder walker
export Build_start_list, Build_add_to_list

proc Build_start_list
  Stack +:= [[Walk_list]]

proc Build_add_to_list
  O1, Stack := delete_last(Stack)
  O2, Stack := delete_last(Stack)
  Stack +:= [O2 + [O1]]
```

custwalk.slm

```
export nothing
```

walker.slm

```
import custwalk
export Object_id_for, Runcall, Walk, Walk_list

func Walk(T) -> Entity
  if T(1) in {Runcall, Walk_list}
    return T(1)(tl(T))
  else
    return eval(T(1), tl(T))
  end

func Runcall(lst) -> Entity
  return lst(1) + as_string(Walk_list(rest(lst)))

func Walk_list(lst) -> list of Entity
  return [all Walk(i) for i in lst]

func Object_id_for(cn, iv) -> Entity
  return `new_'+cn+`(\`'+iv+`\')'

func as_string(lst) -> String
  return `()' when lst = []
  result := `('
  for e in all_but_last(lst)
    result +:= mkstr(e)+`, '
  loop
  result +:= mkstr(last(lst))+`)'
```


identifier.slm

```
import number
export Identifier, new_identifier

Identifier : set of Entity := {}
name : map from Identifier to String

stored_value : map from String to Number

func new_identifier(n : String) -> Identifier
  result := new(Identifier)
  name(result) := n

func value(self : Identifier) -> Number
  result := stored_value(name(self))
  if result = Undefined
    repeat
      put name(self)`?' & get v
    until v in number !number is the built-in numeric type
    result := new_number(mkstr(v))
    stored_value(name(self)) := result
  end
```

number.slm

```
export Number, new_number

Number : set of Entity := {}
slim_value : map from Number to number !number is the built-in numeric type

func new_number(v : String) -> Number
  result := new(Number)
  slim_value(result) := mknum(v)

func display(self : Number) -> Number
  put slim_value(self)
  return self

func divide(self : Number, other : Number) -> Number
  result := new(Number)
  slim_value(result) := slim_value(self) / slim_value(other)

func minus(self : Number, other : Number) -> Number
  result := new(Number)
  slim_value(result) := slim_value(self) - slim_value(other)

func multiply(self : Number, other : Number) -> Number
  result := new(Number)
  slim_value(result) := slim_value(self) * slim_value(other)

func plus(self : Number, other : Number) -> Number
  result := new(Number)
  slim_value(result) := slim_value(self) + slim_value(other)
```

expr.slm

```
import opsys parser builder walker

Parse(Command_line(2)+`.expr' `r')
lines := Walk(Stack(1))

Close(Stdout)
ok := Assign_file(Stdout, Command_line(2)+`.slm', `w')
Terminate_process(Myself, 255) when not ok
put `import identifier, number'
put `'
for l in lines
  put `junk := 'l
loop

Close(Stdout)
ok := Assign_file(Stdout, Command_line(2)+`.lnk', `w')
Terminate_process(Myself, 255) when not ok
put `['Command_line(2)`][identifier number]'

Close(Stdout)

Execute_command(`slim '+Command_line(2))
Delete_file(Command_line(2)+`.slm')
Delete_file(Command_line(2)+`.obj')
Delete_file(Command_line(2)+`.lnk')
Delete_file(Command_line(2)+`.rsp')
```

Appendix 2

code.slm

```
export Code_string

Code_string : String := `'
```

expr.slm

```
import opsys parser builder walker code

Parse(Command_line(2)+`.expr' `r')
junk := Walk(Stack(1))
put Code_string
```


identifier.slm

```
import number unumber
export Identifier, new_identifier

Identifier : set of Entity := {}
name : map from Identifier to String

stored_value : map from String to Number+Unumber

func new_identifier(n : String) -> Identifier
  result := new(Identifier)
  name(result) := n

func value(self : Identifier) -> Entity
  result := stored_value(name(self))
  if result = Undefined
    result := new_unumber(`get')
    code_name(result) := name(self)
    stored_value(name(self)) := result
  end
```

number.slm

```
import code unumber
export Number, new_number

Number : set of Entity := {}
slim_value : map from Number to number !number is the built-in numeric type

func new_number(v : String) -> Number
  result := new(Number)
  slim_value(result) := mknum(v)

func code_for(self : Number) -> String
  return `'

func code_name(self : Number) -> String
  return mkstr(slim_value(self))

func display(self : Number) -> Number
  Code_string +:= `put '+mkstr(slim_value(self))+`\n'
  return self

func divide(self : Number, other : Entity) -> Entity
  return unumber.divide(self, other) when other in Unumber
  result := new(Number)
  slim_value(result) := slim_value(self) / slim_value(other)

func minus(self : Number, other : Entity) -> Entity
  return unumber.minus(self, other) when other in Unumber
  result := new(Number)
  slim_value(result) := slim_value(self) - slim_value(other)

func multiply(self : Number, other : Entity) -> Entity
  return unumber.minus(self, other) when other in Unumber
  result := new(Number)
  slim_value(result) := slim_value(self) * slim_value(other)

func plus(self : Number, other : Entity) -> Entity
  return unumber.minus(self, other) when other in Unumber
  result := new(Number)
  slim_value(result) := slim_value(self) + slim_value(other)
```


unumber.slm

```
import code
export Unumber, new_unumber

Unumber : set of Entity := {}
operator : map from Unumber to String
operand1 : map from Unumber to Entity
operand2 : map from Unumber to Entity
code_name : map from Unumber to String

func new_unumber(op : String, opnd1, opnd2 : Entity) -> Unumber
  result := new(Unumber)
  operator(result) := op
  operand1(result) := opnd1
  operand2(result) := opnd2
  code_name(result) := new_code_name()

func code_for(self : Unumber) -> String
  code_for(self) := `' !only execute this function once
  return `put \`'+code_name(self)+`?\' &\n' +
    `repeat get '+code_name(self)+`: Integer until '+code_name(self)+
      ` /= Undefined'+`\n'
    when operator(self) = `get'
  result := code_for(operand1(self)) + code_for(operand2(self))
  result +:= code_name(self)+` := '
  if operator(self) = `divide'
    result +:= code_name(operand1(self))+`/'+code_name(operand2(self))+`\n'
  elsif operator(self) = `minus'
    result +:= code_name(operand1(self))+`-'+code_name(operand2(self))+`\n'
  elsif operator(self) = `multiply'
    result +:= code_name(operand1(self))+`*'+code_name(operand2(self))+`\n'
  elsif operator(self) = `plus'
    result +:= code_name(operand1(self))+`+'+code_name(operand2(self))+`\n'
  else
    assert False
  end
endf

Name_counter : Integer := 0

func new_code_name -> String
  Name_counter +:= 1
  return `temp'+mkstr(Name_counter)

func display(self : Unumber) -> Unumber
  Code_string +:= code_for(self)+`put '+code_name(self)+`\n'
  return self

func divide(self : Entity, other : Entity) -> Unumber
  return new_unumber(`divide', self, other)

func minus(self : Entity, other : Entity) -> Unumber
  return new_unumber(`minus', self, other)

func multiply(self : Entity, other : Entity) -> Unumber
  return new_unumber(`multiply', self, other)

func plus(self : Entity, other : Entity) -> Unumber
  return new_unumber(`plus', self, other)
```

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.

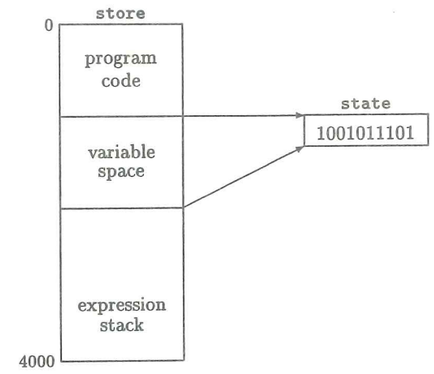

Figure 2: *The abstract machine memory layout.*

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

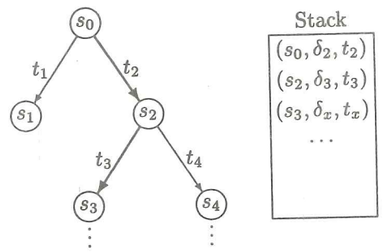

Figure 3: *The stack contains the current path*

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 $0 \ldots 4$ 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 $\mathtt{z}'$, 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 expected the 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.

A Validation Model of the VMTP Transport Level Protocol

HN Roux and PJA de Villiers
Department of Computer Science
University of Stellenbosch, Stellenbosch 7600

Abstract

The implementation of protocols can be speeded up by incorporating formally validated designs in documents that describe new protocols. Unfortunately, this seldom happens in practice. It is proposed that a formally validated model of a protocol be developed before an implementation is attempted. Although this may seem like additional work, this approach can potentially reduce development time because misconceptions are ruled out at an earlier stage. Protocol validation is an established field of research. The techniques are mature enough to allow routine industrial applications. Currently suitable case studies are needed to introduce these techniques to industry. In this paper we describe the development of a validation model of VMTP, a transaction-based protocol that was used as a basis to develop an interprocess communication facility for a microkernel. While the validation of a specific protocol is a one-time event, the development techniques are of more lasting value and are applicable to other protocols.

1 Introduction

The validation of computer protocols has been a field of increasing activity during the past twenty years. Human creativity is required to design effective protocols, but the analysis of protocols can be automated. A vast literature exists on the topic of formal protocol analysis and a few years ago the first book appeared that describes the principles involved [3]. With such a complex topic, however, case studies are essential to introduce newcomers to this fascinating and important area. This paper describes such a case study.

VMTP (Versatile Message Transaction protocol) is a relatively new protocol that was developed as an alternative to TCP[5] or ISO TP4[4]. Local area networks are highly reliable and this is exploited by using group acknowledgements and selective retransmission. VMTP directly supports the client-server model of distributed processing. In this paper we describe the development of an efficient communication mechanism for a microkernel with VMTP as a basis. Some features of VMTP were ignored since they were not needed.

To bridge the gap between the informal description of VMTP in [1] and the implementation, formal models were developed and analysed by using a computer-aided validation technique. We focus on this stage of the development. In Section 2 a brief overview of VMTP is given. Section 3 gives an overview of the development strategy and models for an important component of VMTP are described in Section 4. Examples of correctness properties that were validated are given in Section 5 and we conclude by discussing the experience gained during the project.

2 The VMTP protocol

A brief description of VMTP is included here to provide the necessary background for understanding the validation models. The full protocol description can be found in [1]. The VMTP protocol offers several features to improve the performance of client-server applications. Requests are issued in the form of *message transactions*, which may consist of one or more *packets*. Each packet has a header with fields to identify the transaction to which it belongs and the number of packets in the transaction. The protocol can be divided into the following interacting components: *client support*, *server support* and *management*. Separate models were developed for each component and these separate validated components served as a blueprint for implementation.

The purpose of the client support component is to accept requests from client processes and transmit these requests to servers in a distributed system. The start state (idle state) of the client support component is labelled *Idle* (see Figure 1) to indicate that control is inside some client process. Transactions may consist of up to 32 packets, depending on the amount of data to be transferred. For maximum efficiency, the protocol rules for single packets are extremely simple as can be seen in Figure 1. After accepting all packets of a request message, control reaches the state *Awaiting Response*. The arrival of a single response packet switches control back to the *Idle* state. On the other hand, the first of a sequence of packets in a multi-packet response message transfers control to the state *Receiving Response* where the remaining packets are accepted. Short timeouts are used to trigger selective retransmissions of lost packets, with a longer timeout for outstanding transactions.

The server support component interacts with client support to deliver transactions at their destinations. The start state of this component is labelled *Await Request* to indicate that a new request can be accepted. When the final packet of a request has been received, control switches to a state labelled *Processing Request* to indicate that the server is executing. The response is handled in a similar way as shown in Figure 2. Explicit acknowledgement of transactions is optional and may be requested by clients. However, in most cases the arrival of a response serves as an implicit acknowledgement. A single short packet is used to indicate which packets to retransmit.

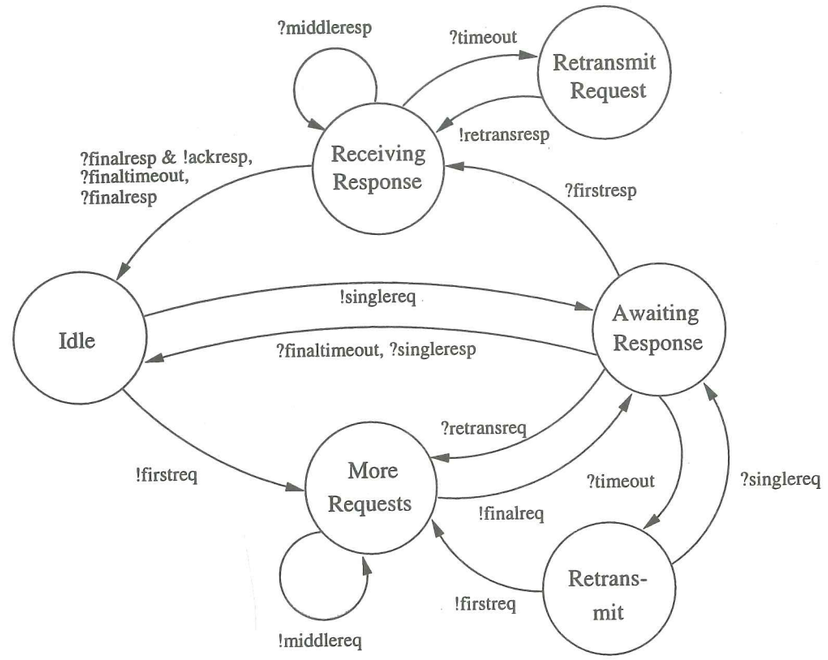

Figure 1: State diagram for VMTP client support

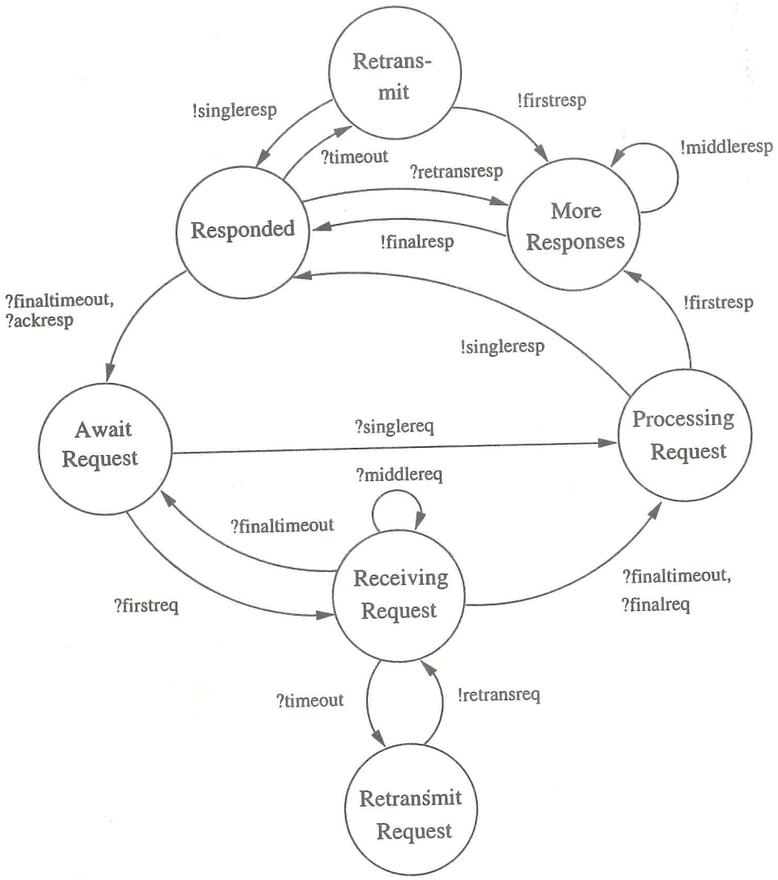

Figure 2: State diagram for VMTP server support

3 Formal specification and validation

Most protocols are published in the form of natural language descriptions and diagrams. The problem is that such informal descriptions are often ambiguous and cannot be analysed for correctness. To address this problem, protocol specification languages were developed. SDL, Estelle and Lotos are well-known examples. A formal specification of a protocol can be seen as an abstract program. The idea is to suppress the implementation detail, but to model the control flow explicitly. Obviously, a formal notation does not prevent the specification of protocols that are incorrect or even impossible to implement. Validation is necessary to ensure correctness.

Formal specification languages were designed before automated validation techniques were feasible for large protocols and unfortunately the well-known specification languages include infinite constructs that cannot be validated efficiently or that may even be impossible to validate in some cases. Consequently, it is necessary to restrict these notations to allow validation. However, many designers of validation tools prefer to experiment with their own notations that are designed for efficiency. The validation language used in Section 4 is such an experimental notation.

The protocol implementation technique proposed in this paper consists of three steps:

- Study an informal description of the protocol and use a formal validation system to build models of all non-trivial mechanisms
- Refine and combine these separate models to produce a validated model of the full protocol
- Use the validated model to derive an implementation

The goal of the first step is to check one's understanding of the protocol description. The rigour of this process can eliminate many potential errors before any code is written. The goal of the second step is to produce a document that describes the protocol precisely. The final model should be shown (by using a computer-aided validation technique) to be free of commonly occurring protocol errors. The final step is to use the formal description of the protocol (and not the original informal description) as a blueprint to develop an efficient implementation. The formal description also serves as documentation and therefore the completed implementation should preserve the structure of the protocol model.

Application of these techniques is not as clear-cut as outlined above. For example, it would be a mistake to complete the formal protocol model without trying out some ideas at the implementation level. It is easy to design a perfectly correct model of a protocol, only to discover that it cannot be implemented efficiently. Simple prototypes of some mechanisms should therefore be constructed during the modelling process. The first goal should be to discover

an elegant design structure. At this stage, validation should be used mainly to check for deadlock because a deadlock usually means that there is something fundamentally wrong with the structure of a design. This should be corrected as early as possible. Once a suitable structure has been developed, one should check whether it can be implemented efficiently. To do this, a skeletal implementation (in the chosen implementation language) should be developed to identify awkward mechanisms as soon as possible. We have found that it works best to refine the model and skeletal implementation simultaneously, one serving as a check against the other.

4 Modelling client and server support

Models are expressed in the experimental validation language ESML. A validation tool that is based on exhaustive state exploration has been developed to check models expressed in ESML. Only those features of ESML needed to understand the models will be explained here. A full description of the language can be found in [2]. ESML is a process oriented language. Concurrent processes communicate via unidirectional channels (A and B in Figure 3) that are accessed via ports (in and out). All processes in ESML are executed concurrently and synchronous message passing is used for synchronisation. A send operation is indicated by "!" and a receive operation by "?". Dijkstra guarded commands are used as control structures. Loops (indicated by the reserved word "DO") execute until all guards are false. Communication commands may be used in the guards of a special "POLL" loop that is repeated until at least one guard is true.

The first model for client and server support ignores multi-packet transactions. After initialisation, two processes *Client* and *Server* repeatedly exchange packets. A variable of type State is used in each process to indicate the current state of the protocol. Two auxiliary variables *msgsent* and *msgreceived* are used to record the occurrence of two important events.

Even this simple model was useful, because it determined a basic structure that could be shown to be deadlock free. It was also shown that all specified events could happen and that all transactions sent were correctly received. The temporal logic CTL is used to express correctness claims. This makes it possible to state properties such as "event α will happen eventually" or "property β will always hold". For example, the CTL formula

$$AG(Client.msgsent \Rightarrow AF(Server.msgreceived))$$

was used to specify that all transactions will eventually be transmitted. Intuitively the formula means that for all execution sequences (signified by the operator AG), when a message is sent, it will eventually (signified by the operator AF) be received. This model generated only 56 unique states.


```
MODEL VMTP;
TYPE Msg = {req, resp};
VAR A, B: Msg; (* channel definitions *)

PROCESS Client(IN in: Msg; OUT out: Msg);
TYPE State = processing, awaitresp;
VAR s: State; msgsent: BOOLEAN;
BEGIN
  (* initialise *)
  s := processing; msgsent := FALSE;
  (* continually react to events *)
  DO s = processing ->
     out!req; s := awaitresp; msgsent := TRUE
  [] s = awaitresp ->
     in?resp; s := processing
  END
END Client;

PROCESS Server(IN in: Msg; OUT out: Msg);
TYPE State = processing, awaitreq;
VAR s: State; msgreceived: BOOLEAN;
BEGIN
  (* initialise *)
  s := awaitreq; msgreceived := FALSE;
  (* continually react to events *)
  DO s = awaitreq ->
     in?req; s := processing; msgreceived := TRUE
  [] s = processing ->
     out!resp; s := awaitreq
  END
END Server;

BEGIN
  Client(A, B); Server(B, A)
END VMTP
```

Figure 3: A high-level model of single packet requests

Various more detailed models were developed, each designed to zoom in on some mechanism by adding detail ignored in earlier models. The behaviour of timeouts were studied by extending the model shown in Figure 3. There are several ways to model timeouts. A special process can be used to represent the "environment". This process emits timeouts as messages. To indicate that a message can be lost (and trigger a timeout) a POLL command can be used as shown below. The POLL will terminate when either a response message arrives or a timeout occurs, which is a natural way to model reality.

```
POLL in?resp -> ...
[] in?timeout -> ...
END
```

Another approach is to couple timeouts to the communication channel which is modelled by a separate process. This process receives messages from its input port and can choose nondeterministically whether to transmit them via its output port or to transmit timeouts instead. Such a channel can be modelled by the ESML code shown below. The POLL command has three guards that may be selected depending on the kind of message that becomes available. A response message may therefore be passed on or a timeout may occur, since any one of the last two guards may be selected nondeterministically.

```
PROCESS Ethernet(IN in: Msg; OUT out: Msg;
BEGIN
  POLL in?req -> out!req
  [] in?resp -> out!resp
  [] in?resp -> out!timeout
  END
END Ethernet;
```

A model that uses the latter approach to model timeouts is shown in the appendix. The advantage of this approach is that it has one fewer process because the environment is not modelled explicitly. This reduces the number of states generated by the model from 7717 to 1248. This is hardly significant for such simple models, but a large number of concurrent processes may cause a severe state explosion when these models are refined. Every technique that can reduce the number of states generated by a model should therefore be exploited.

The final model for client and server support generates about 7 million states. It is much more detailed than the model shown in the appendix, although the same structure could be preserved, even in the implementation.

5 Experience

A number of important correctness claims were validated. These include freedom from deadlock, response to all transmission requests and correct behaviour when packets are lost. It would be a mistake to develop a model of a complete protocol before attempting to validate some correctness claims. Checks to show absence of deadlock are useful from the start. Other useful checks may show that some specified events can never happen. The control logic of the model may be wrong (in which case these "dead" events may well contain errors) or these events may be superfluous because the conditions that guard their execution can never become true. Protocol validation models and prototype implementations were refined simultaneously. It was easy to repeat correctness checks as more detail were added to the models—the refined models were simply validated against the same correctness claims.

Some correctness claims require substantially less computing resources to validate than others. Absence of deadlock is a useful property that is easy to check. An example of a more complex property to validate is that all transactions that represent requests will eventually be transmitted successfully. This property can be specified by a CTL formula of the form $AG(p \Rightarrow AFq)$, where p represents the fact that a transaction has been accepted from a client and q indicates that the specific transaction has been delivered to a server. For the complete model of VMTP 6.9 million states had to be explored to check this property. This required about 30 minutes processing time on a Silicon Graphics Indy workstation.

6 Conclusions and future work

We have described a strategy to implement protocols when no formal description is available. Most protocol descriptions are still published as informal descriptions which, at best, include some diagrams. It is proposed that the informal description should be used to develop a model of the protocol that can be formally validated. This is an effective way to check that the protocol actually works. Many protocols contain errors and it is far less time consuming to find the errors *before* an implementation is attempted. The validated protocol model is then used as documentation to develop an efficient implementation. In other words, it is proposed that informal descriptions should never be used directly to develop implementations. It can only be hoped that in the near future new protocols will be published in formally specified form. This will save time, since a new protocol can be validated once and for all. At the moment, there is no choice: a new protocol for which a validated formal description is not available may contain errors that may go undetected until a product has been in use for quite some time.

The VMTP protocol was selected as a basis to derive an efficient interprocess

communication facility for a microkernel. No formally validated specification was available for VMTP, but a design for the new communication facility was developed and formally validated. Some features of VMTP were not used, but it should be possible to develop a validated model of full VMTP by using the techniques described in this paper.

Appendix: A model of VMTP timeout behaviour

```
MODEL VMTP;
  TYPE Msg = {req, resp, timeout};
  VAR A, B, C, D : Msg;

  PROCESS Client(IN in: Msg; OUT out: Msg);
  TYPE State = processing, awaitresp;
  VAR s: State; msgsent: BOOLEAN;
    ResponseReceived: BOOLEAN; TimeUp: BOOLEAN;
  BEGIN
    s := Processing; msgsent := FALSE;
    responsereceived := FALSE; timeup := FALSE;
    DO s = processing ->
      POLL out!req -> s := awaitresp; msgsent := TRUE
      [] in?resp -> SKIP
      END
    [] s = awaitresp ->
      POLL in?resp -> SKIP; responsereceived := TRUE
      [] in?timeout -> SKIP; timeup := TRUE
      END;
      s := processing
  END
END Client;

PROCESS Server(IN in: Msg; OUT out: Msg);
TYPE State = processing, awaitreq;
VAR s: State; msgreceived: BOOLEAN; responsesent: BOOLEAN;
BEGIN
  s := awaitreq; msgReceived := FALSE;
  responsesend := FALSE;
  DO s = awaitreq ->
     in?req; s := processing; msgreceived := TRUE
  [] s = processing ->
     out!resp; s := awaitreq; responsesent := TRUE
  END
END Server;
```



```
  PROCESS Ethernet(IN in: Msg; OUT out: Msg);
  BEGIN
    DO TRUE ->
      POLL in?req -> out!req
      [] in?resp -> out!timeout
      [] in?resp -> out!resp
      END
    END
  END Ethernet;

BEGIN
  Client(D, A);
  Server(B, C);
  Ethernet(A, B);
  Ethernet(C, D)
END VMTP;

(* CTL formula to claim that when the client sends a
   message it will either be notified that the message
   has been received or a timeout will occur *)

ASSERT AG(Client.msgsent =>
  AF(Client.responsereceived OR Client.timeup))
```

References

[1] D. R. Cheriton. VMTP: Versatile Message Transaction Protocol. RFC 1045, Stanford University, 1988.

[2] P. de Villiers and W. Visser. ESML—A Validation Language for Concurrent Systems. In Judy Bishop, editor, *7-th Southern African Computer Symposium*, pages 59–64, July 1992.

[3] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[4] ISO. Information Processing Systems—Open Systems Interconnection: Basic Reference Model. Standard 7498, International Standardization Organization and International Electrotechnical Committee, 1981.

[5] J. Postel. Transmission Control Protocol. RFC 793, University of Southern California, September 1981.

AUTOMATED NETWORK MANAGEMENT USING ARTIFICIAL INTELLIGENCE

M. Watzenboeck
University of Botswana-Gaborone
E-mail: watzenbo@noka.ub.bw

Abstract

Automating network management is regarded as a prerequisite for further plant automation. Artificial intelligence techniques, such as case based reasoning, topological reasoning and automated learning enable automated network and plant operations and also support tool development for general automation..

1 INTRODUCTION

The approach presented here was initiated by Sumitomo Metals and IBM Japan in 1987. It has achieved a satisfactory level of automated network operations in 1989. Since then the main emphasis lies with tool development for plant operations automation and cooperation enablers.

2 AUTOMATING AUTOMATION

2.1 Case-based Reasoning for Network Automation

The implementation of a computerized helpdesk via case-based reasoning was used a vehicle to achieve the automation of network functions, such as performance monitoring, defective path bypassing and overcoming slowdown effects by buffersize management. Since the AAAI-91 the productivity increase in knowledge engineering by separating knowledge into generic and episodic knowledge is studied worldwide and led among many others to Petrak's VIE-CBR[1].

2.2 Multimedia Front-Ends for Network Management Systems

Shared knowledge of network data enable the most productive use for network management, if skill differences in interpretation through different users are bridged by multimedia support.

Ganesan K. et al.[2] presented the idea of multimedia frontends for networking expert systems on the IEEE Globecom '87 in Tokyo. In CA-World '95 in New Orleans Computer Associates presented the Betaversion of a network management system CA-Unicenter/TNG(The New Generation). J. Lanier's design offers virtual reality(VR) elements and 3D-animations. Applying keyboard, joystick and mouse allows the user to travel into the critical spots of his enterprise guided by a graphic interface. Zooming can reach defective system or network components, advice for repair is given. Traditional network management tools such as HP Open View or Sun-Net Manager can be integrated. The network management part itself is based on the NetDirector of UB Networks For reference see Veitl et al.[3], Damper et al(eds.) on Multimedia Technologies and Future Applications[4],and [5].

2.3 Integrating a Communications Network for Manufacturing Applications

The ESPRIT Project 7096 Computer Integrated Manufacturing and Engineering(CIME) started in 1993 and is based on an open software platform and using standard interfaces(MAP 3.0) . The network management component supports the configuration of the network, monitors its performance, localizes and diagnoses defects and suggests repair actions. Flexible manufacturing is supported by Agile Intelligent Manufacturing Systems(AIMS). Those comprise self-controlling production islands and autonomous-cooperative structures supporting man-machine communication. The peripheral system parts such as CAD, production planning and maintenance are fully integrated[5].

2.4 Group Decision Support and Quality Management

The quality of a product or a service is ensured through quality control meaurements(ISO9001) and guidance(ISO8402). Total quality management changes the focus from the defect-free product to the process. Permanent quality monitoring and improvement becomes an obligation of top management. Process improvements require the participation of all people engaged in the process. The performance of networks is benchmarked against service level agreements. Deviations may necessitate negotiations for new agreements if technical solutions are out of sight. Lincoln[ ] argues: "The parallel between the structures of advanced production plants and Japanese organization is explained by the substitution in both cases of a social for a technological imperative." Greene R.[2] describes the AI based social delivery vehicles suitable for this approach: auxiliary knowledge engineers, application qualification tours, tool courses, group readings and coding sessions in AI and cognition mprovement trainings.

3 SUMITOMO METALS NETWORK AUTOMATION

In 1988 Sumitomo Metals needed to control more than 1000 terminals attached to dual IBM hosts in an SNA environment with response time requirements below 0.8 seconds. NetView is the product family offering SNA Management Services. The used version resides on hosts and supports management tasks, such as configuration, problem and change management, performance monitoring and tuning , accounting and general network operation. All managed objects are connnected to VTAM mostly via NCP. VTAM is the control point for management. NetView monitors the network via Session Awareness(SAW), receipt of VTAM messages, reports from the network components and NetView's command capability The protocol used is the Network Management Transport Vector(NMVT). Filters can be applied to the NMVT messages. In order to support Sumitomo's RYO VTAM a gateway has been built to make the non-SNA devices emulate SNA devices. NMVT does not have such a clean information model as Internet's SNMP and ISO CMIP. Corresponding to their information model but less structured the NMVT carries protocol messages and their permitted values require lengthy search procedures. Alert and Response Time monitoring were the major applications for NMVT. Basic Alert subvector and Generic Alert Subvevtor identify among others alert type, cause of alert and component type. The actual response data is contained in the RTM subvector. Syntax and semantics of the various objects and their attibutes are defined at multiple places and make specifications hard to manage. The availability of products for of all seven SNA-layers was deceisive for the chosen approach.

4 AUTOMATED OPERATIONS USING AI

4.1 A Model for Managing Communication Objects

On the level of the information model no correspondence between SNA's NMTV patchwork and TCP/IP's Management Information Base(MIB) was achieved and the so the ISO standard Abstract Syntax Notation One(ASN.1) was not applied for describing the NMTV protocol. The identification of OPS5(=KnowledgeTool) modules with network management agents allowed dividing the function of network monitoring into logical modules . Automated Network Administation <=> Management Application(Management Agent) <=>Managed Object(Monitoring Agent).

The Management Agent employs a Summarization Monitoring Agent(NetView) to collect and filter the network monitoring information from various Monitoring Agents which are responsible for one or multiple managed objects. All these functions are provided through Netview.

4.2 The Access to Monitoring Agents

Each network addressable unit in a SNA environment is called a node. In a node there is always a Monitoring Agent, which the SNA terminology calls Physical Unit(PU).A Management Agent is a pre-configured listener for the monitored information. The change of pre-selected states in the managed objects triggers events which are forwarded to the listener without an explicit read request. The events are conveyed in terms of event codes and event arguments, similar to error codes of computer programs. The accessible information is modeled as hierarchical objects, the attibutes having complicated data types. The protocol allows searching by filters or browsing through the hierarchy.

5. Case Based Reasoning for Helpdesk Functions

This function is achieved by a 'fusion' of data base and expert system techology. The structure of each LHS in a rule is restricted to three conditions fitting into a mask of 214 bytes each and the RHS is restricted to maximally three actions with maximally 199 bytes, thus allowing the storage of all rules in a relational database. This allows a straightforward implementation of a learning helpdesk through rule updates in the relational rule data base.

6 BIBLIOGRAPHY

[1] Petrak J.: VIE-CBR: Vienna Case-Based Reasoning Tool, Version 1.0, Programmer's and Installation Manual OEFAI-94-34

[2]Ganesan, K. et al.:A Multimedia Front-End for an Expert Network Management System in IEEE Journal, On selected areas in communications, Vol 6, No. 5., June '88

[3] Veitl, M. et al.: Entwicklung multimedialer CBT-Systemw TR-93-4 Austrian Research Institute for Arificail Intelligence.

[4] Watzenboeck, M.: Automated Network Management Using Knowledgetool and NetView, IBM Man. GG24-3435, San Jose, Calif.

[5] COM Software 9/95: CA Unicenter/TNG with virtual reality: "The New Generation"

[6] Malle, K.: Kleinserien rund um die Uhr in: VDI Nachrichten, Produktion Nr. 35, 1.Sept. 1995.

A Framework for Executing Multiple Computational Intelligent Programs

HL Viktor*+ I Cloete+
hlviktor@econ.up.ac.za; ian@cs.sun.ac.za

+*Computer Science Department, University of Stellenbosch, Stellenbosch 7600, South Africa*

**Department of Informatics, University of Pretoria, Pretoria 0002, South Africa*

Abstract

Computational intelligent programs are capable of discovering interesting relationships contained in "raw" data. These programs, including artificial neural networks, set covering algorithms and decision trees, have been successfully used to address a number of real-world problems in, amongst others, the retail, medical, financial and educational fields. A computationally intelligent program can be very effective and useful, given that the learning problems are sufficiently narrowly defined and the data set contains a distribution of attributes favoured by the program.

Many complex real-world problems, however, pose learning problems which cannot effectively be solved by a single program. These problems may be successfully addressed by using a combination of computational intelligent programs. A framework, which combines computational intelligent programs into a computational network, is presented. Employing more than one program potentially leads to more powerful and versatile results.

1 Introduction

Computational intelligent algorithms, including decision trees, set covering algorithms and artificial neural networks, learn classification rules from data. These programs are capable of discovering interesting relationships in the data and usually require little technical knowledge of the programs. This property provides the benefit that these programs are easy to be used by domain experts. Computational intelligent programs have been successfully used to address a number of real-world problems in, amongst others, the medical, financial, agricultural and educational fields [Fu, 1994], [Towell & Shavlik, 1994].

Computational intelligent methods can be very effective if the learning problem is sufficiently narrowly defined and the distribution of the attributes contained in the data set favours the particular program. Many complex real-world problems, however, pose learning problems

which cannot effectively be solved by a single program. The data sets yielded by these problems contain uncertain and/or incomplete data, are generally very large and complex and contains dynamically changing data. In view of this, the development of a computational network that integrate two or more computational paradigms should prove worthwhile. In addition, the different computational intelligent programs offer complimentary advantages which may lead to complimentary results. Integrating more than one program thus potentially leads to more powerful and versatile results.

This paper gives an introductory overview of a framework for executing multiple computational intelligent programs. In Section 2 the computational intelligent programs which we use in the computational network, is introduced. A framework, which combines computational intelligent programs into a computational network, is presented in Section 3. Finally, in Section 4, some conclusions are reached and future extensions are presented.

2 Computational Intelligent Programs

There is a large amount of literature on computational intelligent approaches and methods, including [Clark, 1989], [Fu, 1994] and [Quinlan, 1994]. These methods learn concept descriptions from training data sets. The concept descriptions are subsequently tested on previously unseen test examples to give an estimated accuracy of the concepts learned.

Symbolic methods focus on producing discrete combinations of features, while *subsymbolic* methods adjust continuous, non-linear weighting of their inputs. We consider two symbolic methods, namely set covering algorithms and decision trees. The other two methods combine artificial neural networks, a subsymbolic method, with (a) symbolic rule insertion and (b) symbolic rule extraction.

2.1 Set covering algorithms

Set covering algorithms construct concept descriptions by repeatedly generating conjunctive expressions until all positive instances of a concept are covered or some threshold is reached. One class of covering algorithms, including CN2 [Clark, 1989] as well as BEXA [Theron & Cloete, 1996] construct conjunctions using a general-to-specific search. In this approach, the algorithm start with a general concept description and specialise it in steps until some termination criterion is met. Each conjunction is evaluated according to an error estimate to select the best conjunct for further specialisation.

2.2 Decision trees

A decision tree generates a classifier by means of a structure that is either (a) a *leaf*, indicating a class, or (b) a *decision node* that specifies some test to be carried out on a single attribute value, with one branch and subtree for each possible outcome of the test.

A data row (case) is classified by starting at the root of the tree and moving through it until a leaf is encountered. The decision tree program contains heuristic methods for simplifying the tree; with the aim of producing comprehensible structures without compromising unseen cases. The C4.5 decision tree [Quinlan, 1994] is currently considered to be the state of the art and is used in our computational network.

2.3 Artificial Neural Networks

Artificial neural networks (ANNs) are a class of learning systems that model the human brain. The network consists of a number of weighted units, which can be one or three types: input, hidden or output. Units do only one thing, i.e. they compute a real-numbered output that is a function of real-numbered inputs. Inputs receive the initial numeric attributes from the environment. Hidden units act as links between the inputs and outputs. The outputs correspond to the expected outcome of a training set data row. Artificial neural networks learn the relationships between numeric inputs and outputs by minimising the difference between the expected and actual outputs via weight adaption. Training of the ANN is done by adapting weights via a gradient search. In essence, an ANN performs a nonlinear regression.

2.3.1 Knowledge-based ANNs

In this approach prior knowledge is inserted in the network and subsequently refined by ANN training. In the first step, a set of inference rules that describe the domain knowledge is gathered, usually from domain experts. In this way, the ANN is able to effectively make use of prior knowledge to perform well. Secondly, the knowledge is re-represented in an ANN, and subsequently refined using ANN learning as well as a training data set.

The knowledge defines the topology and weights of the network it creates. The way in which knowledge is re-represented in an ANN is to individually translate each rule into a subnetwork that accurately reproduces the behaviour of the rule. Additional nodes may be introduced to handle disjunctions in the rule set. See [Towell & Shavlik, 1994], [Cloete & Viktor, 1996] for a description of the rule insertion process.

2.3.2 Rules from ANNs

The numerical representation of the attributes as well as the "*black box*" nature of the ANN makes it difficult to determine how a particular decision was reached. Domain experts may be sceptic about the decision reached. If the network produced an interesting discovery, it would be beneficial if this was made explicit. Rule extraction from the ANN attach symbolic meaning to the learning process.

The rule extraction algorithm considers the sum of the weighted inputs to each hidden and output unit. It forms rules by taking the combination of inputs that exceed a threshold as the antecedents and the hidden/output unit as the consequence. The final rule set consists of a combination of these rules which have been reduced using propositional logic and applying sensitivity analysis. See [Viktor *et al*, 1995] for a detailed discussion of our algorithm.

3 Framework

The combination of computation intelligent programs are problem dependant. Therefore, the aims of the proposed framework are to be versatile and dynamically changeable. The framework consists of the following general layers, as depicted in figure 1:

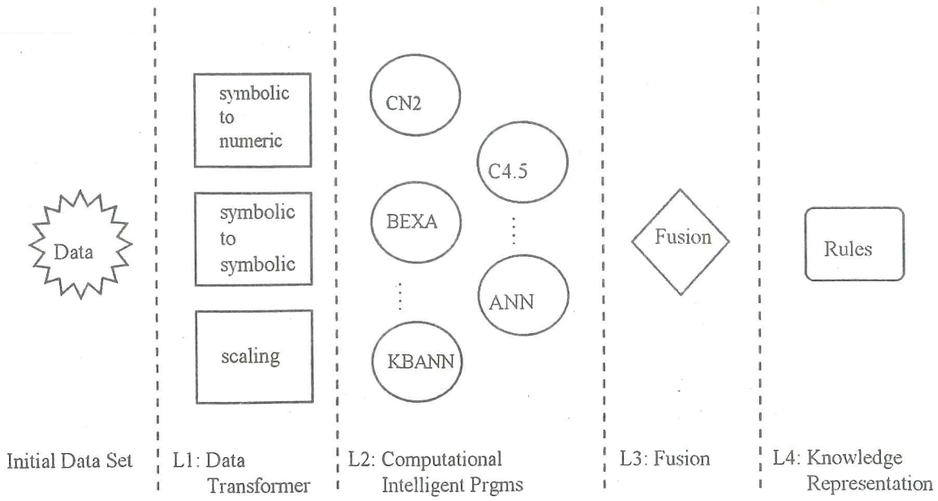

Figure 1: A framework for executing multiple computational intelligent programs

3.1 Data set translation, insertion and selection layer

The first task of this layer is to translate the initial data set to an equivalent representation without any loss of information. The input data representation of the four component programs vary considerably. This layer includes programs to convert data to CN2, BEXA, C4.5 and ANN format. For an ANN, the symbolic data are translated to a numeric representation by the use a scaling algorithm.

Secondly, the selection of the various training data sets for insertion into the computational intelligent programs is addressed. The data set may be partitioned into non-overlapping sets, fully replicated or partially replicated as needed. Again, this selection process is problem dependent. Usually, if the data set is small, wide and the level of noise is high then the data are fully replicated. For example, a noisy tuberçulosis data set consisting of 344 rows and containing 144 inputs yielded the best results when fully replicated [Viktor & Cloete, 1996].

3.2 Individual Computational Intelligent Programs

This layer may consist of a number of sublayers, where the outputs of layer i act as input to layer $i+1$. Each of the individual computational intelligent programs executes autonomously. In addition, the programs have the ability to obtain and use the results of the other programs. The number and type of individual computational intelligent programs are problem dependent. More than one program of a specific type may be executed in parallel. For example, a variety of ANNs, each using a different training function, can be utilised.

3.3 Fusion layer

The fusion layer consists of methods which interpret and combine the results obtained in the previous layer. It also contains procedures to test the accuracy and comprehensibility of the knowledge discovered by the individual components of layer 2. The fusion process is dependant on the fragmentation of the training set. If the training set has been partitioned, the fusion set usually combines the results of layer 2. Replication of data usually require that some selection process is performed. Criteria such as the accuracy of individual rules, the number of examples covered and the comprehensibility of the rules are used to yield an optimum rule set.

3.4 Discovered knowledge

The final layer contains a set of rules which represents the knowledge discovered by the framework. Additional information, including the overall rule set accuracy, rule and attribute relevance, individual rule accuracy and coverage, rule set size and average number of attributes used, is also provided.

4 Conclusion

The ability of a computational intelligent program to find the best solution to a given problem is partially determined by the data set representation. A number of factors, including training-set size and the ability of the program to discover interesting relationships, can mediate the effect of the data representation on the accuracy of the learned concept descriptions. By constructing a framework which takes advantage of each component programs' strengths and data representation, the effect of the data set representation may be further minimised.

Current research includes the experimental evaluation and refinement of our framework. Some of the results which we obtained by considering a tuberculosis data set are discussed in [Viktor & Cloete, 1996]. The development of a framework for executing computational intelligent programs is a promising approach to machine learning. Since it combines the strengths of various computational intelligent programs, it has the potential to address real-world problems which could not previously be solved by single methods.

References

[Clark, 1989] P Clark and T Niblett, 1989. The CN2 Induction Algorithm. Machine Learning, 3.

[Cloete & Viktor, 1996] I Cloete and HL Viktor, 1996. Inserting Domain Knowledge into Artificial Neural Networks, Technical Report ANN/RI/01/96, Department of Computer Science, University of Stellenbosch, Stellenbosch, South Africa.

[Fu, 1994] LM Fu, 1994. Neural Networks in Computational Intelligence, McGraw-Hill.

[Matheus et al, 1993] Matheus CJ, Chan PK & Piatetsky-Shapiro G. 1993. Systems for Knowledge Discovery in Databases. IEEE Transactions on Knowledge Data Engineering, 5(6), p904-913.

[Quinlan, 1994] JR Quinlan, 1994. C4.5: Programs for Machine Learning, Morgan Kaufmann.

[Towell & Shavlik, 1994] GG Towell and JW Shavlik, 1994. Refining Symbolic Knowledge using Neural Networks, Machine Learning, (editors RS Michalski and G Tecuci), 4, Morgan Kaufmann.

[Theron & Cloete, 1996] H Theron and I Cloete, 1996. BEXA: A Covering Algorithm for learning Propositional Concept Descriptions, Machine Learning, 14, p321-331.

[Viktor *et al*, 1995] HL Viktor, AP Engelbrecht and I Cloete, 1995. Reduction of Symbolic Rules from Artificial Neural Networks using Sensitivity Analysis, IEEE ICNN'95, Perth, Australia.

[Viktor & Cloete, 1996] HL Viktor and I Cloete. 1996. Extracting Knowledge from Tuberculosis Data, *to appear in* Methods for Informatics in Medicine.

A Script-based prototype for Dynamic Deadlock Avoidance

C N Blewett* and G J Erwin#

* Dept. of Accounting and Finance (Business Information Systems Section), University of Natal, King George V Ave., Durban 4001, South Africa, BLEWETT@BIS.UND.AC.ZA

\# Dept. of Accountancy (Business Information Systems Section), University of Durban-Westville, Private Bag X54001, Durban 4000, South Africa, ERWIN@IS.UDW.AC.ZA

Abstract

Expert systems apply Artificial Intelligence (AI) techniques to an application area, aiming (usually) to mimic the behaviour of a human expert. However, there are some AI techniques which can be used to improve the internal performance of an existing application, not necessarily currently performed by a human. In this paper, we present further research and results of EAGLE (**E**xternal **A**dvisor for **G**ranting **L**ocks **E**xpertly), an expert system advisor for the lock manager in a database system. By matching lock event sequences received from the lock manager against stored scripted deadlock sequences, EAGLE is able to identify unfolding deadlock sequences. By using this Dynamic Deadlock Avoidance (DDA) approach, EAGLE is able to avoid deadlocks before they occur. Currently, no ideal solution exists to the deadlock problem. Solutions vary in terms of the number of waits for access to resources and the number of deadlock occurrences. By utilising AI techniques, DDA offers a new way of treating the deadlock problem. In this paper we describe the design of EAGLE and present the results, in terms of the number of waits and number of deadlock occurrences, occurring with DDA compared with Deadlock Detection and Resolution.

Keywords

database system, deadlock, expert system, scripts, knowledge representation, machine learning, plan recognition

1. Introduction

"Deadlock is a situation in a resource allocation system in which two or more processes are in a simultaneous wait state, each one waiting for one of the others to release a resource before it can proceed." Deadlock may occur within database, communication and distributed systems using locking as a concurrency control strategy [3]. The deadlock state and its treatment are well-described. See, for example, [1], [5], [7], [10], [12] and [22].

Blewett and Erwin [6] describe *four* categories of deadlock treatment: *viz. ignore deadlock, deadlock detection and resolution, deadlock prevention* and *deadlock avoidance*.
Detection with recovery allows the system to enter a deadlock state and then recover from it. Prevention ensures that a deadlock will never happen. Avoidance should also ensure no deadlocks, but, can only do that by using *a priori* data about the locking activities of transactions (resource consumers) [3]..

The dynamic deadlock avoidance (DDA) approach described in this paper does *not* require pre-stored knowledge of lock request sequences, but attempts to notice potential deadlocks based on experience accumulated from observations of the previous locking event sequences which led to deadlock. DDA allows deadlock to occur, records the conditions which led to the deadlock, then watches for similar, deadlock-producing conditions in future.

DDA treats the deadlock problem as a *plan recognition* issue [4], rather than as a *problem resolution* issue. The DDA approach attempts to identify whether the "goal" of the running "plan" of current locking event sequences is deadlock. When this goal is recognised, DDA OBJECTs to further locking activities which could lead to a future deadlock.

In the following sections we first describe the overall design of our DDA-based prototype, called EAGLE (**E**xternal **A**dvisor for **G**ranting **L**ocks **E**xpertly), incorporating script-based knowledge representation [18], [19] and [20], in a centralised resource allocation (database) system. We then discuss the implementation of the EAGLE prototype, EAGLE's components, and test runs under simulated conditions within a database system. We describe the sets of test data used to assess the impact of EAGLE on the occurrence of deadlock under those simulated conditions and present some preliminary results from those test runs.