

Using Model Checking to Debug and Verify Electronic Trade Protocols

Robert Zimmer, Brunel University, Uxbridge, Middx. UB8 3PH

Keywords: E-commerce, Formal methods, Trade protocols

Abstract: Model checking is a semi-automated formal verification technique that has proved to be remarkably successful in verifying and debugging micro-electronic (and software) designs. In this paper, it is shown (using a case study) how model checking can be applied to e-commerce, providing assurances of safety of trade procedures and protocols before they are put into practice.

1. Introduction

Safe and effective use of business-to-business electronic commerce requires that the trade procedures and expectations of the companies involved in an exchange be well understood and modelled. This modelling will allow possible conflicts to be discovered and resolved in negotiation, before they occur in practice. Over the last several years, a technique called *model checking* has proved remarkably successful in doing exactly this kind of reasoning about micro-electronic hardware (and software) design. For example, we have used model checking to find bugs in published circuit designs; and model checking has been used to find errors in cache coherence protocols for multiprocessors [Zimmer and MacDolad,1993]. Some work has been done applying these techniques to interaction protocols for e-commerce [Heinze *et al*, 1996]. In this paper, we argue that the same techniques are applicable to inter-organisational electronic commerce, and that, in fact, some microelectronic design debugging and verification software can be used to debug and verify trade procedures. We believe this can lead to safer and more powerful forms of electronic interaction in a commercial setting, as it has in an electronic design setting..

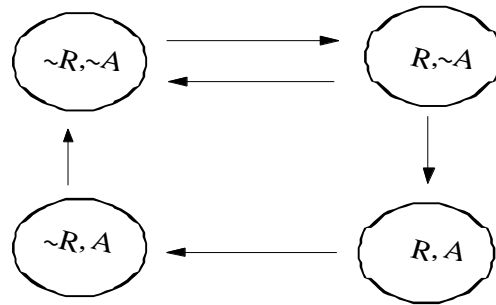
2. An Example State Space

Model Checking, as it was originally set out, involves computations on finite Kripke models. Kripke models (from a systems modelling point of view) are just directed graphs, or finite state machines without the inputs, that layout the possible state changes of a system. As an example, which will be followed throughout most of this paper, consider the following mistaken simple handshaking protocol:

There are two signals an outgoing Request, R, and an incoming Acknowledge, A.

The protocol works as expected except, sometimes, a request will disappear without its having been acknowledged.

A state space (or Kripke model) for this system is given below:



Each state is an assignment of values to each of the variables. The state in the upper left hand corner, for example, represents the system with both the Request and Acknowledge Signals off. In model checking, the R and A are thought of as Boolean variables that are either true or false in each state: for example in the top right-hand state, R is true and A is false. The arrows represent state transitions: in a state transition the value of some of the variables may change. If the protocol were working as expected, only the clockwise arrows would be in the state space. The one anti-clockwise arrow represents a request signal turning off before it is acknowledged.

The idea of model checking is to take State Spaces, such as the one above, and ask questions about them. The questions are of two forms:

(I) Does state X have property p?

or

(II) Which states of the space have property p?

The properties themselves are either about the truth values at a given state, or about the truth values at possible future states. For example, one query may ask at which states is R true, and another may ask from which states is there a path to a state in which R is true (that is, in what states is it true that R may be true in the future). These conditions about future states are called *temporal conditions*.

Neither of the properties mentioned above is terribly interesting, but temporal logic enables the expression of very useful requirements, especially for control structures. The logics used in model checking are all species of propositional temporal logic; the logic that is most frequently used for model checking is called **Computation Tree Logic** or **CTL**.

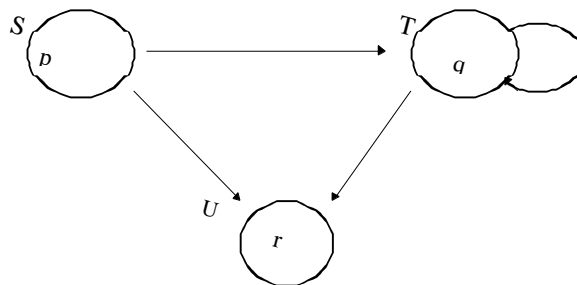
3. Computation Tree Logic

Like all propositional temporal logics, the formulae in Computation Tree Logic include the usual propositional ones — that is, as well as variables, the connectives "and", "or", "not", and "implies" are allowed — and some temporal operators.

The operators and their meanings will be defined at the same time. The semantic (meaning) relation is between states and formulae. The relation is expressed by $s \models \phi$, where s is a state and ϕ is a formula, and is to be interpreted as saying that ϕ is true in state s. Then the definition of \models is given by:

$s \models \sim$	if and only if s does not satisfy .
$s \models \text{ or}$	if and only if s satisfies at least one of and .
$s \models \&$	if and only if s satisfies at both of and .
$s \models \Rightarrow$	if and only if $s \models (\sim \text{ or })$.
$s \models E()$	if and only if in some immediate successor to s , holds. That is, it is possible for to hold in the next state.
$s \models A()$	if and only if in all immediate successors to s , holds. That is, it is necessary that will hold in the next state.
$s \models (Au)$	on all paths from s , holds at all states until the first state in which holds (although is not assumed to ever hold).
$s \models (Eu)$	on some maximal ¹ path from s , holds at all states until the first state in which holds (although, again, is not assumed to ever hold).
$s \models (EU)$	on some path from s , holds at all states until the first state in which holds (here, is assumed to eventually hold)
$s \models (AU)$	on all paths from s , holds at all states until the first state in which holds (must eventually hold on every maximal path)

The differences between the small and large u's are subtle. To get a feeling for it consider the following Kripke model:



The letters outside of the circles are the names of the states, and the letters inside the circles represent properties that are true in the states.

There are infinitely many maximal paths from S: there is the length one path going straight to U; there are paths going from S to T, staying in T for some number of cycles, and then moving to U, and there is the infinite path that goes to T and stays there (by repeatedly going around the loop).

¹ A maximal path is either a path that comes to a dead-end or an infinite path.

On all of those paths, q remains true until r is true. Therefore

$$S \models q \text{ AU } r$$

However on the infinite path r is never true, so

$$S \models \sim (q \text{ AU } r)$$

Meanwhile there are some paths in which r is eventually true (in fact, on almost all of them) so:

$$S \models q \text{ EU } r$$

To see how this language would be used in practice, some conditions for checking that the state space given above is a good implementation of a handshaking protocol will be considered. Below are listed some properties and their formalisations:

(I) If a request is made it **will** continue to be made until it is acknowledged

$$R \rightarrow R \text{ AU } A$$

(II) If a request is made it **may** continue to be made until it is acknowledged

$$R \rightarrow R \text{ EU } A$$

(III) If nothing is being acknowledged, then nothing will be acknowledged until a request is made

$$\sim A \rightarrow \sim A \text{ AU } R$$

(IV) In the state immediately after a request is acknowledged, the request will stop

$$A \ \& \ R \rightarrow A(\sim R)$$

(Note that the A for All should not be confused with the A for Acknowledge)

And so on. The most difficult part of model checking is deciding what properties are sufficient and necessary as a specification. This specification is certainly not complete.

4. Model Checking State Spaces

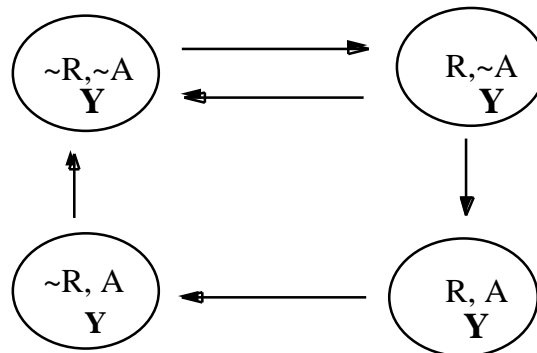
Checking simple conditions, like the propositional ones or $E(\)$, is straightforward and quite clearly computable: the algorithm involves looking ahead at most one time step. For each of the more complicated temporal constructs, the

model checking algorithm is an iterative searching technique. The main theorem that makes the method work is that these iterative techniques all finish after a number of steps bounded by the length of the longest non-looping path in the state space. Therefore, the whole process can be automated: one keeps performing iterations until the same result is reached twice. This is called a *fixed point*. The theorem states that a fixed point will be reached in bounded time.

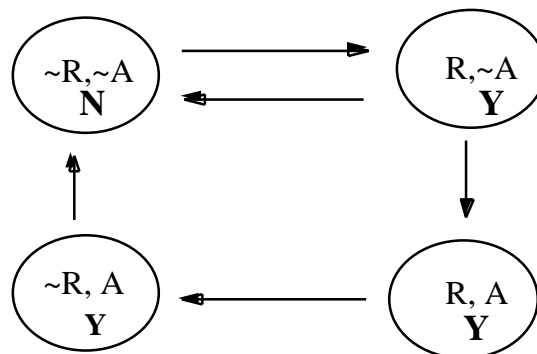
1.1.1 Example 1:

As a first example, the following computation will be done: what states in the state satisfy $R \text{ Au } A$. Recall that this means that from that state every path satisfies R until such a time when it satisfies A .

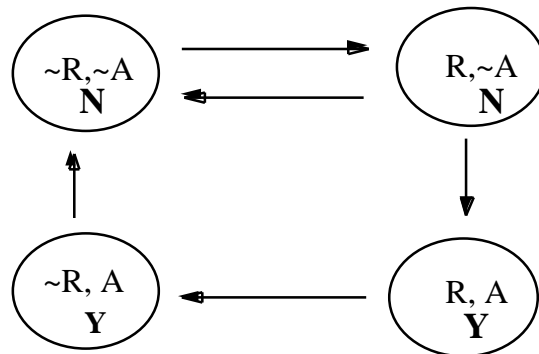
To compute an Au every state is initially labelled with a Y (for Yes), and, in each iteration of the search algorithm, some or none of the Y 's may become N 's. The Y 's label the initial model because the condition is a universal one (a forAll). The underlying intuition is that it is initially assumed that each state satisfies the condition, and the process is to keep searching until one bad path is found. As soon as the bad path is found, the state from which it started can never satisfy the forAll condition, and a Y permanently becomes an N . When no states change in an iteration, the algorithm is finished.



This is the starting point. Now, the zeroth iteration: the first thing to consider is what would make $R \text{ Au } A$ false without any searching. If both A and R were false, then it would be impossible for R to be true until A was. So the zeroth iteration changes the Y to N in any state in which both variables are false.



From here on, the idea is to turn a Y to an N in state s if one of the places you can get to in one move from s has an N in it, and A is false in s . Looking at the space above, the lower-left and the upper-right both point to an N, and the lower-left satisfies A. The upper right therefore changes:



This opens up one more state to consider: the lower-left now points to an N. However, since that state satisfies A, nothing changes in this iteration. Therefore the algorithm stops, and it is now known that the bottom two states satisfy the condition $(R \text{ Au } A)$ and the top two states do not.

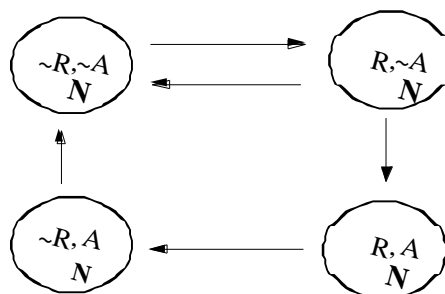
Now to consider requirement I of the specification for the protocol:

(I) $R \rightarrow R \text{ Au } A$

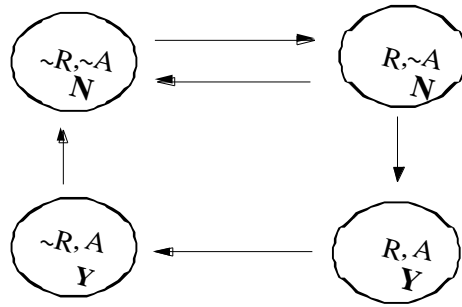
This condition is satisfied in every state except the upper right: where R is true but $(R \text{ Au } A)$ is not. Therefore, not surprisingly, this state space fails the specification. Another thing to note is that not only has the model checker told us the condition is false, but it also provides a counter-example. This makes model checking as valuable for debugging as it is for verification.

1.1.2 Example 2:

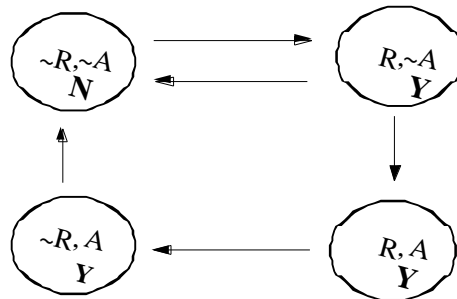
Consider the property $R \text{ Eu } A$. Since the property is an existence property we start with everything labelled with N's and change them to Y's as soon as an appropriate path is found.



This is the starting point. For the zeroth iteration, we think of what would make $R \text{ Eu } A$ true without any searching. The answer is: if A is true.



From here on: an N gets turned to a Y in state s if from s a state with a Y in it can be reached in one move, and R is true in s . In this case, the upper right hand corner points to a state with a Y in it and R is true in that state. Therefore after one iteration the model becomes:



And now the upper left-hand state points to a Y. However, since R is false in that state, nothing changes. and the algorithm halts.

Now consider the second specification condition:

(II) $R \rightarrow R \text{ Eu } A$

This condition is true, since there is no state in which R is true and R Eu A isn't.

1.2 On the Complexity of Model Checking:

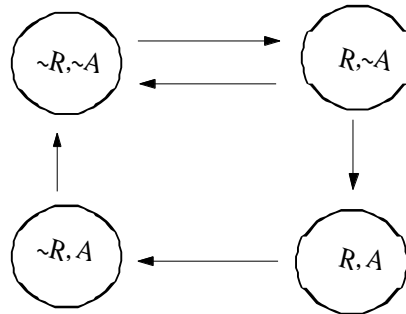
This can be an effective way of verifying conditions of systems, but it can be unwieldy

- The state graph is exponential in the size of the system description with an attendant time and space penalty in model checking. This is called the *state explosion problem*
- Even with quite a small state graph, checking a CTL formula can involve a considerable amount of search.

Where this technique really gets its power is from a new representation of the state transition graph as a boolean function. The representation is called an Ordered Binary Decision Diagram (BDD) [Bryant 86, 92]. The CTL formulae can also be represented in the same way. Since the operations performed in checking a CTL formula against a transition relation can be expressed as operations on BDDs, the model checking process can often be performed efficiently with such a representation. The first step along this path of representation is to view the state space as a boolean function, as will be explained in the next section.

5. From State Spaces to Boolean Expressions

The easiest way to explain this process is by way of an example. So consider once again the state space:



This is now considered to be a system with four boolean variables: R , A , R' , and A' . R and A represent variables in the present state, while R' and A' represent variables in a next state. Each transition (arrow) in the state space is modelled as a conjunction (an and) of these variables. For example the bottom arrow is, which is a transition from a state in which both R and A are true to one in which R is false and A is true, is modelled as: $R \& A \& \sim R' \& A'$. The whole graph is then a disjunction (an or) of these conjunctions. So in this example the boolean expression for the state space is:

$$B = \begin{array}{l} \sim R \& \sim A \& R' \& \sim A' \\ R \& A \& \sim R' \& A' \\ \sim R \& A \& \sim R' \& \sim A' \\ R \& \sim A \& \sim R' \& \sim A' \end{array} + \begin{array}{l} R \& \sim A \& \sim R' \& \sim A' \\ R \& \sim A \& R' \& A' \end{array} + \begin{array}{l} R \& \sim A \& R' \& A' \\ R \& \sim A \& \sim R' \& \sim A' \end{array} + \begin{array}{l} R \& \sim A \& R' \& A' \\ R \& \sim A \& \sim R' \& \sim A' \end{array}$$

5.1 Computations Using Boolean Expressions

Now, suppose we wanted to compute the states that can possibly have both Request and Acknowledge false in the next state, that is the states that satisfy $E(\sim A \& \sim R)$. We can do this as a purely boolean calculation as follows. In general if we want to compute $E(p)$ for some formula p , we need only to use the following formula:

$$E(p) = B \& p'$$

(where p' is the same as p with all of the variables affixed with dashes)

This works because what this says is that the B relation holds (so it is a transition in the state space) and that the p relation will hold of the target.

The calculation is given below:

$$([\sim R \& \sim A \& R' \& \sim A' + R \& \sim A \& \sim R' \& \sim A' + R \& \sim A \& R' \& A' + R \& A \& \sim R' \& A' + \sim R \& A \& \sim R' \& \sim A'] \& (\sim A' \& \sim R') =$$

$$([\sim R \& \sim A \& R' \& \sim A' + R \& \sim A \& \sim R' \& \sim A' + R \& \sim A \& R' \& A' + R \& A \& \sim R' \& A' + \sim R \& A \& \sim R' \& \sim A'] \& \sim A' \& \sim R' =$$

$$(\sim R \& \sim A \& R' \& \sim A' + R \& \sim A \& \sim R' \& \sim A' + \sim R \& A \& \sim R' \& \sim A') \& \sim R' =$$

$$R \& \sim A \& \sim R' \& \sim A' + \sim R \& A \& \sim R' \& \sim A'$$

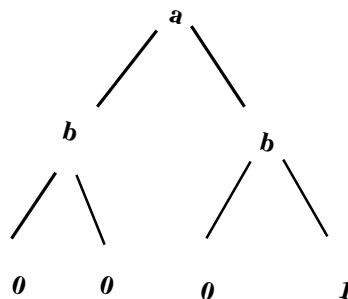
Therefore there are two states that satisfy the formula: one with R and $\sim A$ (the upper-right hand corner) and the one with $\sim R$ and A (the lower-left hand corner). And it is easy to see by inspection that this is right.

All the CTL expressions generate this kind of boolean calculation. The complex temporal operators generate calculations of (either least or greatest) fixed points of boolean functionals. This is too mathematically sophisticated a process to be described here, but conceptually it is a boolean formalisation of the reasoning that was used on the model checking of state spaces. See [McMillan 93] for a more detailed explanation.

Operating on boolean expressions is not necessarily efficient, but the efficiency of these calculations is frequently dramatically increased by moving to ordered binary decision diagrams (BDDs).

6. Reduced Ordered Binary Decision Diagrams

An unordered binary decision diagram is simply a tree form of a truth table. Consider, as a first example, the expression $a \& b$. The binary decision diagram is given by:



As you go down the tree, left means false and right means true. In this tree, for example, to find out the truth value of the function when a is true and b is false, start at the top, go down the right branch (corresponding to 'a' being true), and then at the junction, go down the left branch (corresponding to 'b' being false). Diagrams like these were first introduced by Akers in 1978.

These diagrams were greatly refined by Bryant in 1986. Bryant's improvement was to reduce the diagrams using three reduction operations:

- (1) Join all the 0's and join all the 1's at the bottom of the tree
- (2) Join nodes that do the same thing in the tree
- (3) Remove redundant tests

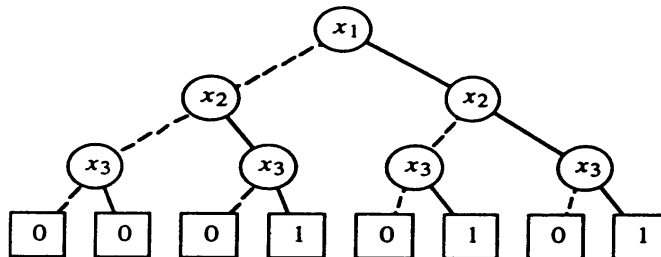
To understand the way these work, consider the following example is taken from Bryant 1992:

The function is

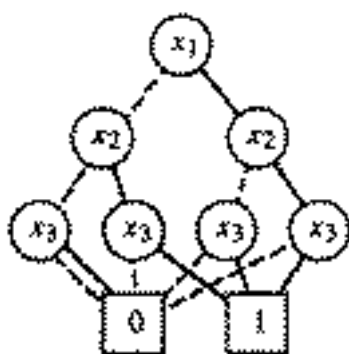
$$F = \sim x_1 x_2 x_3 + x_1 \sim x_2 x_3 + x_1 x_2 \sim x_3$$

The binary decision diagram is:

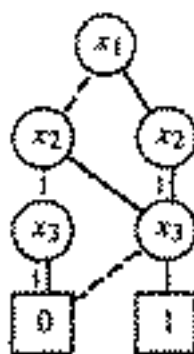
x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



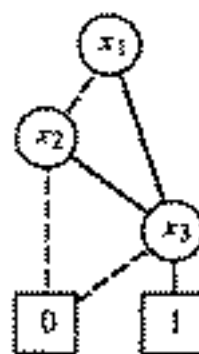
And after the reductions it is:



A). Duplicate Terminals



B). Duplicate Nonterminals



C). Redundant Tests

In this example, the move from A to B entails noticing that the three rightmost x_3 nodes in BDD A all have arrows leading to the same places. They are therefore conglomerated as one node. The move from B to C entails noticing that from the rightmost x_2 in B both arrows lead to the same place: this implies that the x_2 test is unnecessary since the same behaviour is given by both results. The same holds for the leftmost x_3 node. Therefore these two nodes are removed.

The boolean function represented by the final BDD is:

$$G = \sim x_1 x_2 x_3 + x_1 x_3$$

which is equivalent to F.

6.1 The use of a BDD

In the last section we saw how CTL model checking can be expressed as boolean operations. Bryant has shown how to perform all of the standard operations using Reduced BDDs. This is frequently a very efficient method in time (and especially) in space of performing these operations. To each boolean operation, there corresponds an operation on the associated BDDs that is polynomial in the size of the BDDs.

6.2 The size of a BDD

BDDs can provide a compact representation of boolean functions. However, the size of a BDD representation for a boolean function is strongly dependent on the variable ordering. Ideally the optimal variable ordering would be computed automatically. This is computationally infeasible. Thus we must use a heuristic method to determine a reasonable variable ordering. Various techniques have been proposed for determining a good variable ordering.

The size of a BDD can be estimated by viewing the BDD as a sequential processor [Bryant 92] that inputs a bit string sequentially and outputs a single bit when the entire bit string is input. It takes bits in according to the variable ordering — each bit corresponds to a variable in the function being represented. Whenever a bit is input this defines the function computed over the rest of the string. The number of distinct functions that can result from a given sequence of variables is the number of nodes at the corresponding level in the BDD. Thus if we can *estimate* the number of distinct functions, we can define a simple linear search algorithm to determine a variable ordering which is likely to be reasonable.

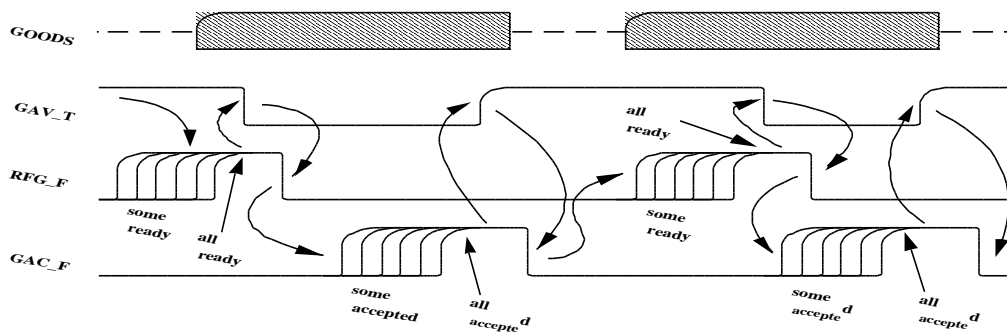
This is quite easy to do. We need to be able to estimate the number of distinct functions in a set of function representations. We simply select some at random and evaluate each them on a set of random arguments. If the number of function representations is known (and it is), then from a sample of such evaluations we can estimate the number of distinct functions represented. In addition we perform the obvious tests on the function representations.

7. Specification of a Multi-party Trading Procedure

In this section, we present a contractual scenario, in which a single seller interacts with multiple purchasers: the parties all agree that delivery will happen if and when all purchasers can take delivery of goods. This may, for example, be the procedure for a business that has a high delivery overhead, and wants to avoid unnecessary trips. The scenario is based on an IEC (International Electrotechnical Commission) standard 625-1 1979 interface protocol for programmable measuring instruments. The protocol includes a 3-wire handshake, which is intended to synchronise the transmission of data bytes over a bus. The handshake is designed to allow a single source to synchronise with a number of acceptors.

The trade procedure is intended to operate as follows: the seller can indicate that he has goods available for delivery (GAV). The purchasers can indicate that they are ready to take goods—that is, they request goods—(RFG) and can also indicate when they have received goods (GAC). The required sequence of events for the transfer of goods from the seller to the purchasers is as follows: GAV, RFG, GAC are initially false (that is GAV_F, RFG_F, GAC_F) and remain false until all purchasers are ready to receive goods, at which time RFG becomes true (RFG_T). The seller may then assert GAV_T and RFG becomes false. When all purchasers have accepted a specified quantity of goods, GAC becomes true and only then may GAV become false. Finally the purchasers may set GAC back to false and the cycle may repeat.

This sequence is illustrated in Figure 1 below:



8. Verification of the Trade Procedure

One condition, a form of liveness, that we wish to check is that from every state the trade procedure can eventually return to its initial state. In other words, we wish to establish that the transaction between the seller and any of the purchasers will eventually terminate. The condition is expressed below (S_0 is the initial state).

$$S_0 \models G (EF (snr \ A (pnr(1) \ pnr(2) \ \dots \ pnr(n))))).$$

Some safety conditions we have also checked are:

(i) The seller should assert GAV_T only when all purchasers are ready—all purchasers should be in state pr. In other words, we wish to establish whether there is a situation where goods are available for delivery on the seller's part but the transaction cannot be realised because some purchaser cannot accept delivery. The condition is expressed below:

$$S_0 \models G ((\neg \text{SellerGAV_T} \ \wedge \ \text{E SellerGAV_T}) \ (\text{pr}(1) \ \text{pr}(2) \ \dots \ \text{pr}(n))).$$

(ii) Once GAV is true, it should remain true until all purchasers have accepted goods—all purchasers should have entered state pf. In other words, we want to establish whether it is possible once the transaction has been entered for the seller to retract the goods before they are delivered (to cancel his part of the bargain). The condition is expressed below:

$$S0 \models G(\text{SellerGAV_T} \wedge (\text{SellerGAV_T} \wedge \text{AU}(\text{pf}(1) \wedge \text{pf}(2) \wedge \dots \wedge \text{pf}(n)))).$$

(iii) No purchaser should accept goods until GAV is true—i.e. none should enter ps until GAV is true. In other words we want to establish that there is no situation where a purchaser's obligation to pay for goods is activated before such goods are available and delivered to him. So we require:

$$\forall i: 1 \leq i \leq n, S0 \models G(\neg \text{ps}(i) \wedge \text{E} \text{ps}(i) \wedge \text{SellerGAV_T}).$$

(iv) No purchaser should accept more than one lot of goods while GAV is true—i.e. none should enter ps more than once (this is what was stipulated in our trade procedure), so we require:

$$\forall i: 1 \leq i \leq n, S0 \models G(\text{ps}(i) \wedge \text{A}(\neg \text{ps}(i) \wedge (\neg \text{ps}(i) \wedge \neg \text{SellerGAV_T}))).$$

These liveness and safety conditions among others were checked by our system, and all hold for the defined trade procedure.

9. Conclusions and Future Work

Business-to-business electronic commerce would greatly benefit from a system that could model and evaluate trade procedures before they are put into practice. This will enable misunderstandings and potential mistakes to be found before they become actual and expensive problems. This paper has reported a technique that performs much the same task for micro-electronic design. That these techniques are applicable in an electronic commerce setting has been demonstrated by following a simple multi-party trade procedure through the major steps in the modelling and verification procedure.

10. References

- S. B. Akers, *Binary Decision Diagrams*, IEEE Transactions on Computers, C-27, 6 (Aug.), pp. 509-516
- M. C. Browne, E. M. Clarke, O. Grumberg, *Reasoning about Networks with Many Identical Finite State Processes*, Information and Computation 81, 1989, pp 13-31.
- R. E. Bryant, *Graph based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, C-35, 6 (Aug.), pp. 677-691
- R. E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams*, Carnegie Mellon Technical Report CMU-CS-92-160, 1992
- E.M.Clarke, E.A. Emerson and A.P. Sistla, *Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications*, ACM Trans Programm Lang Systems 8, No 2, 1986, pp 244-263.
- Nevin Heintze, J.D. Tygar, Jeannette Wing, and H. Chi Wong, "Model Checking Electronic Commerce Protocols, Proceedings of the Second USENIX Workshop on Electronic Commerce 1996. Pp. 147-164.
- K.L. McMillan, *Symbolic Model Checking*, Kluwer 1993.
- R Zimmer and A MacDonald, "The Algebra of System Design: A Petri Net Model of Modular Composition", IEEE International Symposium on Circuits and Systems, 1993