Under consideration for publication in Formal Aspects of Computing

Using SPIN to Analyse the Tree Identification phase of the IEEE 1394 High Performance Serial Bus (FireWire) Protocol

M. Calder and A. Miller

Department of Computing Science
University of Glasgow
Glasgow, Scotland.

Abstract. We describe how the Tree Identification phase of the IEEE 1394 High Performance Serial Bus (FireWire) protocol is modelled in Promela and verified using SPIN. The verification of arbitrary system configurations is discussed.

Keywords: model checking, SPIN, formal verification

1. Introduction

Model checking enables us to analyse communication protocols by exhaustive inspection of reachable composite system states in a finite state machine representation of the system. The SPIN model checker [Hol93] has been widely used in numerous verification case-studies and industrial software-controlled systems [LS97, CGM$^+$97, CM01b, Hol99, HS99, HP00, MS00]. In this paper we use SPIN to analyse the Tree Identification phase of the FireWire protocol. Model checking is by no means the "push-button" technology that it is rumoured to be. It is crucial that the abstraction made of the system is both true to the original system and high-level enough to allow exhaustive exploration of the entire state-space. It is possible, for example, through inefficient abstraction, to completely fail to prove any properties of a system due to state-space explosion. In this paper the authors apply their previous experience with the SPIN model checker (see for example [CM01b]) to model and analyse the FireWire protocol. By applying the optimisation and model-generation techniques described in [CM01b], we are able to create a model with a tractable state-space and

Correspondence and offprint requests to: Alice Miller, Department of Computing Science, University of Glasgow. e-mail: alice@dcs.gla.ac.uk

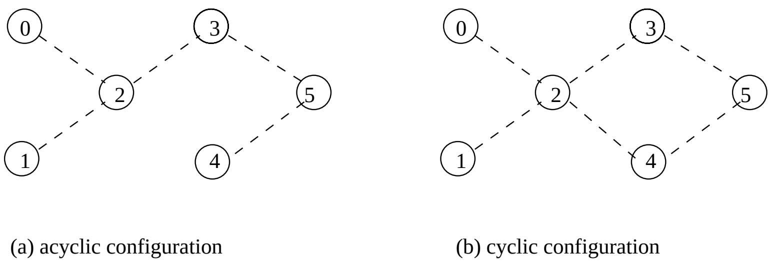

Fig. 1. Network configurations

perform analysis upon it – so allowing for a fair comparison with the other verification methods discussed within this volume.

In general, model checking can only be applied to to a *particular* network (following results established in [AK86]). In some specific instances, when the topology is very simple (e.g. a ring or a star), model checking can be applied to parameterised systems [EN95, EN98, GS92, ID96]. The main goal of this paper is to demonstrate how, for a fixed number of nodes, *for any configuration*, SPIN can be used to verify certain properties– in which case a different model is required for each network configuration. As one of our primary interests is that of the generalisation of model checking results to configurations of any size, we have also considered a slightly different model to that of the IEEE model [IEE95, IEE00]. In this model the protocol is slightly modified: each incoming request is acknowledged before processing to the next, whereas in the IEEE model, all incoming requests are acknowledged collectively. The reason for our decision to also analyse this modified protocol is that in this model, a configuration of N nodes will systematically degenerate to a configuration of $N-1$ nodes (unlike the IEEE model). This degenerative behaviour is interesting in itself, and naturally leads to an "inductive" approach to reasoning about configurations on any number of nodes. We discuss how, for this model, for any system of N nodes with a star topology, certain properties hold.

In this paper we create models of all connected networks of six nodes where messages are passed in accordance with (i) the Tree Identify protocol (TIP) and (ii) the modified Tree Identify protocol (MTIP). Six nodes are more than sufficient to explore all possible behaviours of the protocol and a small enough number to ensure tractability of model checking runs. Examples of the type of system that are considered are the acyclic and cyclic configurations on 6 nodes given in figure 1.
Analysis takes the form of proving, for a network on 6 nodes, a set of properties including:

- If the network is cyclic an error will be reported,

and if the network is acyclic:

- a unique leader is always elected
- any node can be elected as leader.

Note that we have not considered the "force root" scenario (in which a device – for which a *force_root* flag is set to *true* – may influence its own chances of becoming root by waiting for some time before sending a parent request, even if it is already possible to proceed).

A further goal is to explore the possibility of extending model checking results that apply to all configurations on N nodes, to configurations on more than N nodes. In this paper we give an initial solution to this problem for a certain subclass of configurations (star topologies) of the MTIP.

In section 2 we give a brief overview of Promela and SPIN. The general approach to model checking and an abstract representation of the node process are discussed in section 3. In sections 4 and 5 we describe our models relating to the TIP and the MTIP and the properties that we wish to prove. Techniques used to optimise the Promela code and to generate Promela models automatically are described in section 6. In section 7 we give the results of verification for both of our models. In section 8 we give an abstraction which enables us to extend results to fixed configurations of N processes, for large N, when there is a star topology (for the MTIP). A method for converting an abstracted (MTIP) model for (star topologies of) fixed N to a model that describes the behaviour of systems with a star topology of any size N is also described. In section 9 we compare our approach to other approaches outlined in the workshop proceedings [MRS01] and in section 10 we evaluate our solution by considering the "points to address" posed in said proceedings. Finally, in section 11 we describe our conclusions and further work.

2. Promela and SPIN

Here we give an overview of Promela and SPIN. More details of the search algorithms and parameters used by SPIN are contained in [CM01a].

Promela, *Pro*cess *me*ta *la*nguage [Hol93, Hol97a], is a high-level, state-based, language for modelling communicating, concurrent processes. It is an imperative, C-like language with additional constructs for non-determinism, asynchronous and synchronous communication, dynamic process creation, and mobile connections, i.e. communication channels can be passed along other communication channels. Thus, the language is very powerful and expressive. In section 2.1 we give some of the features of Promela that are necessary for the Promela description of our system, as described in section 4.

SPIN is a bespoke model checker for Promela and provides several reasoning mechanisms: assertion checking, acceptance/progress states and cycle detection and satisfaction of linear temporal logic (LTL) formulae. In sections 2.2 and 2.3 we briefly describe SPIN's search algorithm and temporal reasoning in SPIN. Some of the additional parameters used in SPIN verification that we refer to later in this paper are described in section 2.4.

2.1. Promela

As well as the usual integer data types (bit, bool, short, int, unsigned) and arrays, Promela allows the use of an additional datatype, *mtype*. Such a data type can only be used following an mytpe declaration which is used to define symbolic names of numeric constants. For any verification model there is only one such declaration allowed. The definition

```
mtype = {be_my_parent,be_my_child,ack}
```

is functionally equivalent to the sequence of macro definitions:

```
#define  be_my_parent 1
#define  be_my_child 2
#define  ack 3
```

If an mtype declaration is present, the keyword `mtype` can be used as a data type, to introduce variables that obtain their values from the range that was declared. This data type can also be used inside channel declarations (see below), for specifying the type of message fields.

Communication between active processes takes place via *channels*. Channels are declared using the keyword `chan`, either locally or globally and (usually) store messages in first-in first-out order. Asynchronous communication takes place via channels of length > 0 (and synchronous communication on channels of length 0). All communication in our system is asynchronous and channels have length 1. The channel declaration

```
chan chanout = [1] of { mtype }
```

declares that the channel of name *chanout* can store 1 message of type mtype. We have chosen to have channel size equal to 1 to keep our model as simple as possible.

A *send* statement on a buffered channel is by default executable in every global system state where the target channel is non-full. The statement *chanout!be_my_parent* writes the message *be_my_parent* to *chanout* provided the channel is not full.

If *chanout* is declared as above, a *receive* statement *chanout?m_message*, where *m_message* is an mtype variable, is executable in every global system state where *chanout* is non-empty and the (oldest) message is removed from the channel.

An *assert* statement is similar to the predefined condition statement `skip` in the sense that it is always executable and has no other effect on the state of the system than to change the control-state of the process that executes it. However, most importantly it can trap violations of simple safety properties (via "assertion violations") during verification and simulation runs with SPIN.

Process behaviour is declared via a *proctype* construct and then instantiated via a *run* operator during the initialisation process (*init*) or from within another process. Alternatively a process may be initiated with the prefix *active* that can be used at the time of declaration (we do not use this method here). Declarations for local variables and message channels appear within the proctype declaration. Proctype declarations may be parameterised. For example, the following declares a proctype `node` with one formal parameter *selfid*.

```
proctype node(byte selfid)
{ . . .}
```

Any instantiation of a `node` process (via a `run` operator) will involve assigning a value to the parameter, for example: `run node(0)`.

Sometimes (for the purposes of verification) we need to keep track of the *pid* (process identifier) of a process. In this case we associate a variable with the corresponding run statement. For example, if it is necessary to access the pid of a `node` process with parameter `0`, but not of a `node` process with parameter `1`, this may be achieved via the corresponding run operations:

```
p0=run node(0);
run node(1)
```

An *inline* definition plays the same role as that of a function or procedure in an imperative language, such as C. Any such definition must be global and appear before its first use. The body of an inline is copied into the body of a proctype at each point of invocation (or call). An inline call can appear wherever a statement can appear. For example, the `converter` inline definition below takes a pair of process ids (corresponding to the respective process *selfid* parameters, and not to be confused with the pids of the processes, as described above) and converts them to a pair of channels, *chanin* and *chanout*. Notice that the inline statement is specific to the particular configuration of the system, and is generated automatically per configuration (see section 6). This `converter` statement corresponds to the cyclic configuration of figure 1(b).

```
inline converter(id1,id2,chanin,chanout)
/*takes a pair of ids and finds the corresponding in and out channels */
/*need to change per configuration*/
{if
 ::(id1==0)->assert(id2==2);chanin=twozero;chanout=zerotwo
 ::(id1==1)->assert(id2==2);chanin=twoone;chanout=onetwo
 ::(id1==2)->assert((id2==0)||(id2==1)||(id2==3)||(id2==4));
   if
   ::(id2==0)->chanin=zerotwo;chanout=twozero
   ::(id2==1)->chanin=onetwo;chanout=twoone
   ::(id2==3)->chanin=threetwo;chanout=twothree
   ::(id2==4)->chanin=fourtwo;chanout=twofour
   fi
 ::(id1==3)->assert((id2==2)||(id2==5));
   if
   ::(id2==2)->chanin=twothree;chanout=threetwo
   ::(id2==5)->chanin=fivethree;chanout=threefive
   fi
 ::(id1==4)->assert((id2==2)||(id2==5));
    if
    ::(id2==2)->chanin=twofour;chanout=fourtwo
    ::(id2==5)->chanin=fivefour;chanout=fourfive
    fi
 ::(id1==5)->assert((id2==3)||(id2==4));
   if
   ::(id2==3)->chanin=threefive;chanout=fivethree
   ::(id2==4)->chanin=fourfive;chanout=fivefour
   fi
 fi}
```

Finally, a process at an unexecutable statement or false guard (essentially the same in Promela) is blocked; thus Promela implements a form of busy waiting.

2.2. On-the-fly Depth-first Search

In order to perform verification on a model, SPIN constructs an automaton representing the global behaviour of the concurrent system. The automaton has an associated initial state f_0, a finite set of states S and a finite set of transitions T such that T is a set of pairs (s_1, s_2), where $s_1, s_2 \in S$. Each transition of the automaton corresponds to the execution of a specific atomic statement within one of the (concurrent) processes. The automaton can be easily represented by a graph (a *state-graph*) in which the nodes correspond to the states in S and directed edges correspond to the transitions in T.

A basic depth-first search (to check for deadlock, assertion violations etc.) explores the state-graph starting from the initial state f_0, successively progressing along the edges of the graph and back-tracking when a previously visited state is reached, until an error is found – or until the entire search space has been explored.

2.3. Temporal Reasoning in SPIN

As well as enabling a search of the state-space to check for deadlock, assertion violations etc., SPIN allows the checking of the satisfaction of an LTL formula over all execution paths. The mechanism for doing this is via *never claims* – processes which describe *undesirable* behaviour, and Büchi automata – automata that accept a system execution if and only if that execution forces it to pass through one or more of its accepting states infinitely often. Full details of never claims and Büchi automata are given in [Hol97a, GPVW95, MP90]. Here, we give a brief overview of the mechanisms involved and a description of how they have been employed.

Standard LTL formulae are constructed from a set of atomic propositions, the standard Boolean operators ($\neg$, $\wedge$ and $\vee$), and the temporal operators $[]$ (always), $\langle\rangle$ (eventually), $\circ$ (next) and U ((strong) until). (Note that the use of partial order reduction (see below) precludes the use of the next operator during SPIN verification.) Propositions include process control such as $p@label$ meaning process p is at label $label$.

When SPIN is used to verify an LTL property one must first use SPIN's LTL converter which translates LTL formulae into Promela syntax. This translation is a *never-claim* and encodes the Büchi acceptance condition. During a SPIN verification, this never-claim is converted to a Büchi automaton. Another Büchi automaton, consisting of the the synchronous product of the LTS corresponding to the concurrent system (model) and the Büchi automaton corresponding to the never-claim, is constructed. Thus a different Büchi automaton is constructed for each never-claim. A depth-first search explores the state-graph associated with this (new) Büchi automaton.

If the original LTL formula f does not hold, the depth-first search will "catch" at least one execution sequence for which $\neg f$ is true. If f has the form $[]p$, (that is f is a *safety* property), this sequence will contain an *acceptance state* at which $\neg p$ is true. In this case the never-claim is said to *complete*. Alternatively, If f has the form $\langle\rangle p$, (that is f is a *liveness* property), the sequence will contain a cycle which can be repeated infinitely often, throughout which $\neg p$ is true. In this case the never-claim is said to contain an *acceptance cycle*. In either case the never claim is said to be *matched*.

When using SPIN's LTL converter it is possible to check whether a given property holds for *All Executions* or for *No Executions*. A universal quantifier is implicit in the beginning of all LTL formulas and so, to check an LTL property it is natural, therefore, to choose the *All Executions* option. It is not possible to check that a given property (p say) holds for *some state* along *some execution path* using LTL alone. This type of property is verified using SPIN by showing that "$\langle\rangle p$ holds for *No Executions*" is violated (or equivalently "$\neg\langle\rangle p$ holds for *All Executions*" is violated).

2.4. Parameters and Further Options used in SPIN verification

When performing verification with SPIN, there are three parameters that need to be set. These are *Physical Memory Available*, *Estimated State Space size* and *Maximum Search Depth*. The meaning of the first of these is clear, and the second controls the size of the state-storage hash table. The *Maximum Search Depth* parameter (MSD) determines the size of the *search-stack*, where the states in the current search are stored. If an exhaustive search is required, this parameter must be set to a value that is greater than any path explored in the search. However if a search is performed to find a single counter-example (see for example property 3 in section 5), this parameter can be set to the smallest value required to *trap* an error path. As the search-stack is responsible for a large proportion of the total memory requirement of a verification the verifier must ensure that the value of MSD is kept to as small a value as possible. In addition, if comparisons are to be made with other model checkers, for example, the value of MSD should be taken into account. Note that in section 7 we give the memory requirements for state storage only.

Partial order reduction (POR) [Pel96b, Pel96a] is a technique that is used to diminish the time and memory requirements when model checking concurrent processes. It is based on the observation that execution sequences (or "traces") of a concurrent program can be divided into equivalence classes whose members are indistinguishable with respect to a property that is to be checked. By ensuring that at least one trace from each equivalence class is executed during a reduced search, the use of POR ensures that redundant work is not performed and that the truth (or otherwise) of a property is preserved. When performing SPIN verification, POR is applied by default.

Compression [Hol97b] is a method by which each individual state is encoded in a more efficient way. The total memory required for state storage is thus reduced. We apply compression for the verification of all of our properties.

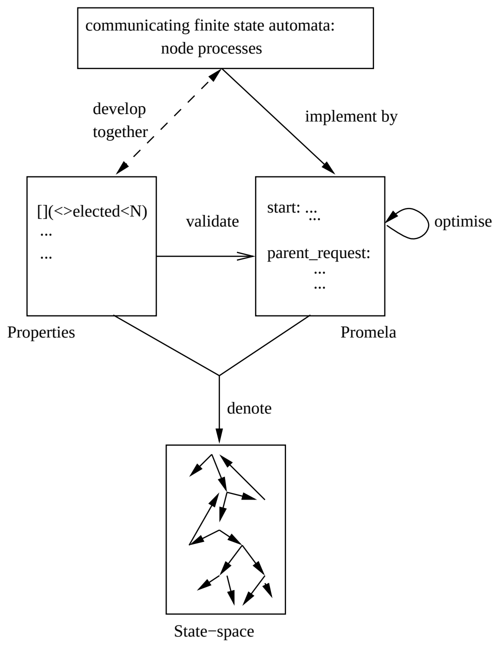

Fig. 2. Overall Approach

Weak Fairness [God96, Pel96a, Bos99] ensures that any process that has a transition that remains enabled will eventually execute it. The algorithm is based on a variant of Choeka's flag algorithm [Cho74] and involves the construction of an extended state-space consisting of N copies of the original state-space (where N is the number of processes). Clearly the additional memory and time requirements of such an algorithm are great and therefore its use should be avoided where possible (see section 5).

Other state-space reduction options available with SPIN include *Minimised Automaton Encoding* [VW86, Hol97b, VB96, HP99] and *Supertrace* (or *Bitstate hashing*) [Hol90, Hol98]. We do not discuss these methods here.

3. The Approach and Node process

In this section we give an overview of the approach taken to analyse the Tree identify protocol (TIP) and a modified version of the TIP (MTIP), using model checking, and provide a (simplified) abstract automaton describing node behaviour in each case.

3.1. Approach

The approach taken is illustrated in figure 2. Each individual stage is described in detail in subsequent sections. Our starting point is the automata and properties (the top and left hand side of figure 2). Neither need to be *complete* specifications; this is a virtue of the approach as it allows us to exclude irrelevant implementation details from our model. The Promela description on the right hand side of figure 2 is regarded as the implementation; a crucial step is therefore validation of the implementation. This is done by checking satisfaction of the properties, using SPIN. In order to avoid state-space explosion, the Promela code must be optimised.

3.2. The Node Process

Figure 3 gives a diagrammatic representation of the automaton for each node process (note the full implementation is somewhat more complicated) in the TIP case. States to the left of the *start* state represent

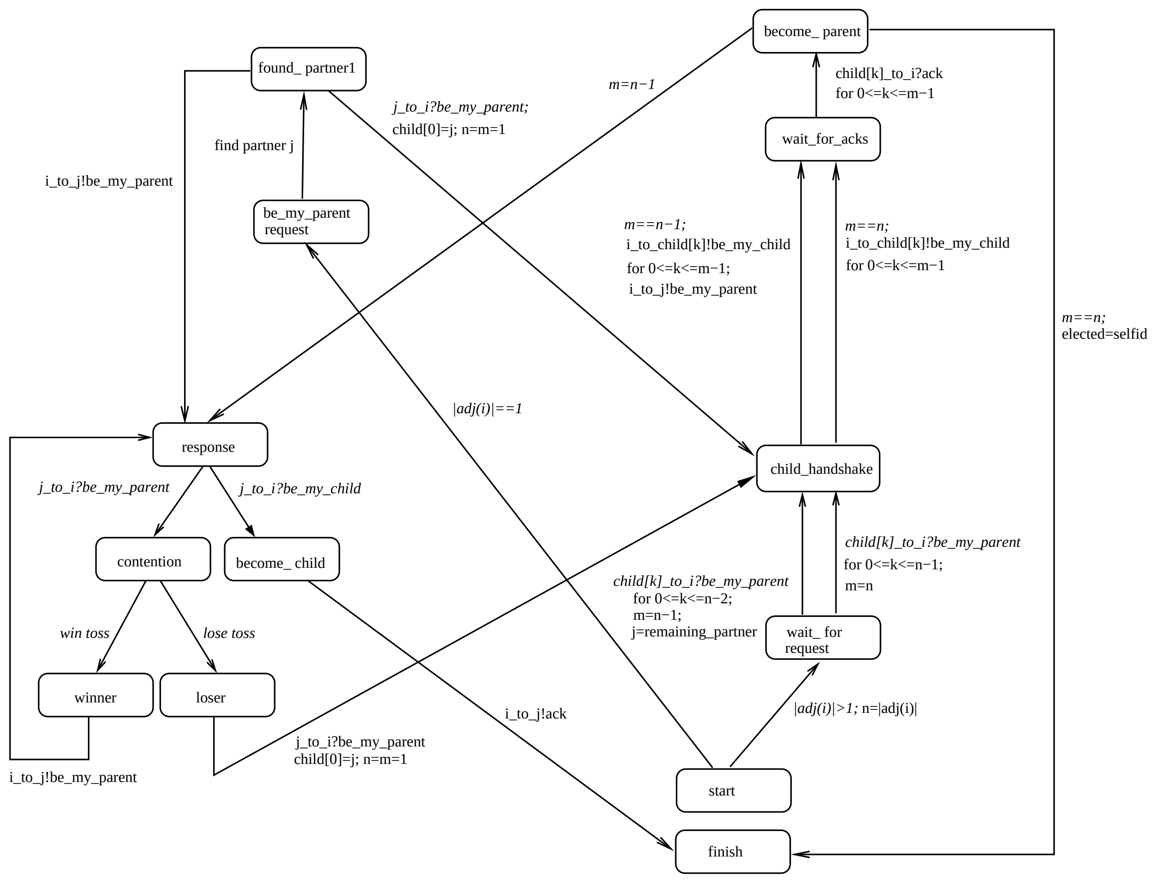

Fig. 3. Finite state automaton representing node i behaviour (TIP)

leaf behaviour and those to the right represent *non-leaf* behaviour. Events label transitions. Events that represent *actions* of the process are given in plain font, whereas *conditions* or *guards* are represented in italics. Events/guards are separated by semicolons. Note that a *read* (*write*) event can be seen as both an action and a condition (it is only executable if the associated channel is full (empty)). Events of the form *channel?message* (or *channel!message*) can be translated as "if the channel is full (empty), read message from (write message to) the channel – otherwise take another branch". An unitalicised event of the form channel?message (or channel!message) on the other hand can be translated as "when the channel is full (empty), read message from (write message to) the channel". In the automaton the variable i refers to the current node, i.e. *selfid*. The variable j is a (free) variable which ranges over possible adjacent nodes. The guard

$$child[k]\_to\_i?be\_my\_parent \ \text{ for } 0 \leq k \leq n-1$$

can be transtated as: "if *be_my_parent* requests are received from n neighbouring nodes, then assign these nodes $child[0], child[1], \ldots, child[n-1]$ and remove messages".

Similarly, figure 4 gives a diagrammatic representation of the automaton for each node process in the MTIP case. States to the left of the *start* state represent *child* behaviour and those to the right represent *parent* behaviour. Note that, in this model each incoming request is acknowledged before the next is processed, unlike the TIP model. Also in this case non-leaves have their children assigned in an incremental way, returning to the *start* state after each assignation. Every time a child is assigned to process i, the value of $|children(i)|$ increases by 1. Eventually the left-hand branch from *start* may be taken. Also, in this version, once a child has been assigned to a node, the node is no longer considered to be connected to the child. Thus it is easier in this case to find remaining unassigned nodes: simply by checking connections. Otherwise, the two models are very similar.

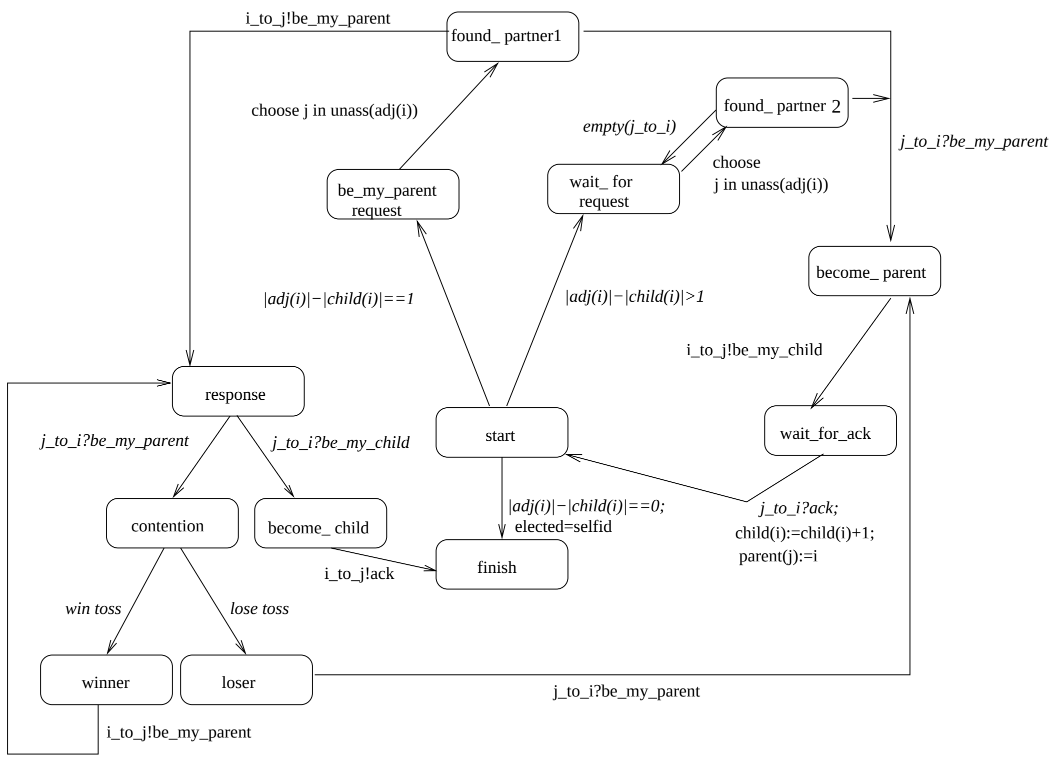

Fig. 4. Finite state automaton representing node *i* behaviour (MTIP)

To implement communication we associate a pair of channels for each pair of processes. The channels associated with *nodeA* and *nodeB* say are *nodeA_to_nodeB* and *nodeB_to_nodeA*. Each channel has capacity for at most one message of type *mtype*.

In both cases, the states *found_partner1* and *found_partner2* are states at which a partner has been selected. The numeric part of these state labels is simply to differentiate between them.

Note that, if a node forms part of a loop in a cyclic configuration (node 5 in figure 1(b) say), then (in the TIP model), upon reaching the *wait_for_request* state, no *be_my_parent* will ever be received. Therefore the node will remain at the *wait_for_request* state indefinitely. In the MTIP model, the node will cycle between the *wait_for_request* state and the *found_partner2* states indefinitely. This is an important observation, and provides a means for loop detection (i.e. a cycle within the node configuration), which is discussed in section 5.

In both models we have forced root contention to be resolved (one of the processes must *lose* and the other *win*). On returning to the *response* state the winner will not be able to revisit the *contention* state because no relevant *be_my_parent* message will be received. The reason for modelling contention in this way is that in model checking all feasible paths are explored. Any infinite path (cycle in the state-graph) will be reported as an error. If contention was allowed to remain unresolved such an infinite path would exist and an error would be reported, preventing any useful results being obtained. A different solution would have been to restrict the number of times that a given contention could remain unresolved to 10 say. However, this would only serve to dramatically increase the size of the state-space without being any more realistic.

4. The Node Process in Promela

Unless otherwise specified, we discuss here the Promela code associated with the TIP model (see figure 3). The code associated with the MTIP model will, of course, be slightly different. Each node process is an instantiation of the (parameterised) proctype *node* declared thus:

```
proctype node (byte selfid)
```

As an example of the Promela code, we give here the code associated with the *contention* state (this is the same in both the TIP and MTIP models). Originally we used a direct analogy of the "timeout" approach to contention resolution via the non-deterministic allocation of timeout constants to each process. However, this soon proved to be impractical. Although in a real-life situation one can see that contention would eventually be resolved (the probability of contention remaining unresolved for an infinite length of time is zero), there is no explicit way of handling probability within Promela. Therefore, to avoid infinitely long paths along which contention is unresolved, we abstract away from this approach and force one of the contending processes to (non-deterministically) have its "be_my_parent" request accepted.

Note that N denotes the number of processes and $self\_in$ and $self\_out$ denote the channels $partner\_to\_self$ and $self\_to\_partner$ respectively. Also note the use of assert statements within the code (see section 2). These are used particularly at points when entering a new state, and when reading and writing to communication channels. They are invaluable for debugging and form an essential part of the verification process.

```
contention:
atomic{
 assert((message==nullmessage)&&(counter==N)&&(partnerid!=N));
 if
 ::selfid<partnerid->counter=selfid
 ::else->counter=partnerid
 fi;
 if
 ::(toss[counter]==0)->toss[counter]++; counter=N;goto winner/*win toss*/
 ::else->assert(toss[counter]==1);toss[counter]=0;counter=N;goto loser /*lose toss*/
 fi};
```

Code associated with each individual *state* of figure 3 (*start, be_my_parent_request, found_partner1* etc.) is contained within an atomic statement. This is to ensure that these states correspond, as far as possible, to the actual *visited states* of the state-graph (see section 2.2).

When a new configuration is to be modelled the set of relevant channels must be declared. In addition the *adj* and *connect.to* arrays must be reinitialised. The former records how many neighbours each node has and the latter records the particular connections that exist (or are still undecided in the MTIP case). Thus if node 0 has 1 and 2 as its neighbours $adj[0]$ is set to 2 and $connect[0].to[1]$ and $connect[0].to[2]$ are both set to 1. These settings are made automatically (see section 6).

5. Analysis

SPIN enables us to verify a system via assertion checking, acceptance/progress states and cycle detection and satisfaction of linear temporal logic (LTL) formulae. The properties that we aim to verify (with associated LTL) are given below. We note that

1. In order to show that a particular configuration is acyclic we show that no process can be waiting for a "be_my_parent" request for an infinitely long time. To do this for the TIP model we show that no process remains at the *wait_for_request* state for an infinitely long time. For the MTIP model we show that no process cycles between the states *wait_for_request* and *found_partner2* for an infinitely long time.
2. Properties 2,3 and 4 assume that the network is acyclic. They are the same for both the TIP and MTIP models.
3. Properties 1,3 and 4 relate to a particular process i, where $0 \leq i < N$, where N is the number of processes. A verification must be performed for each value of i.

Property 1 *Process i will not wait for a "be_my_parent" request for an infinitely long time.*
That is $[](\langle\rangle\neg p)$ where p is $(node[proci]@wait\_for\_request)$ (TIP model) or $((node[proci]@wait\_for\_request) \vee (node[proci]@found\_partner2))$ (MTIP model).

Property 2 *A leader will always be elected.*
That is $(\langle\rangle q)$ where q is $\neg(elected == N)$.

Property 3 *It is possible for process i to be elected leader.*
That is $\neg\langle\rangle r$ is violated, where r is $(elected == i)$.

Property 4 *Only one process will be elected leader.*

That is $[](p \rightarrow ([]p))$ where p is $(elected == i)$.

6. Optimisation and the Use of Scripts

Our recent experience using SPIN [CM01b] enables us to *optimise* the code effectively to reduce the search-space. This involves resetting all variables to their initial values (*reinitialisation*) after use, and removing all references to variables that are not required for the current verification (*refinement*).

Reinitialisation is a similar process to that involved in SPIN's inbuilt *dead-variable elimination* optimization. However, as discussed in [CM01b], this default optimization sometimes results in an *increase* in the size of the state-space. Our manual reinitialisation avoids this problem.

As an example of *refinement*, the $finished$ array used in property 2(b) (see section 7 below) can be removed from the model during the verification of all other properties. The removal of the array results in a reduction of the size of the state-vector (for example, from 208 bytes to 200 during the verification of property 1) and a corresponding reduction in memory required for state-storage. The *slicing* algorithm available with the new release of SPIN [DH99, MT00] appears to refine Promela code in a similar way. As we generate our models automatically, we prefer to create code that is already refined, rather than use the slicing algorithm on unrefined code. However the slicing algorithm is invaluable for alerting the user to refinements that may have been missed previously. (For example, the slicing algorithm warned us that the *leader* variable is never used during the verification of property 1, and so all references to it may be removed for this verification.)

It would be extremely time-consuming (and error-prone) to rewrite our model for each configuration of processes and each property to be verified. Therefore we make extensive use of Perl scripts to generate our models automatically from a template file together with a data file containing an array representing the process configuration. In addition, a Perl script was used to generate model checking runs for all 6 feasible acyclic configurations.

When performing investigations on a system of N nodes, when N is large (see section 8), the use of such scripts was essential. The generation of the properties and connection information alone required automation.

The results obtained in the following section all apply to optimised code. An example of a MTIP model generated for 6 node processes, with the configuration of figure 1(a), to verify property 2 (see section 5) is given in the appendix. (The equivalent TIP model can be found on our website at [CM].)

7. Analysis results

There are many feasible configurations on 6 nodes, our aim is to check the properties for both the TIP and MTIP models in each case. The configurations correspond to connected graphs which can be constructed by the addition of one additional vertex to a tree on 5 vertices (in the spirit of FireWire, in which nodes are added to, or removed from, an acyclic network of nodes). There are 93 such configurations (there are 3 trees on 5 vertices, and 31 possible sets of neighbours for the additional vertex in each case), but we do not know at this stage how many *non-isomorphic* configurations there are. We do know, however, that there are only 6 non-isomorphic acyclic examples, and our properties can be verified for each of these. To check that property 1 is sufficient to check for loops (in the network), we have limited ourselves to the cyclic configuration of figure 1(b). In all cases the appropriate model is generated automatically (see section 6). The *physical memory available* and *estimated state space size* are set to 256 and (the default value of) 500 respectively and POR and *compression* are used throughout. In tables 1 and 2 we give the verification results obtained for each property (denoted by "prop") for the TIP and MTIP models respectively. In each case we include the configuration under consideration (figure 1 ((a) or (b))), and the value of a representative node (i) if appropriate. The use of weak-fairness (or otherwise) is denoted by a $\surd$ or $\times$ in column 2 (see section 2.4 for details). Time and memory requirements of verification clearly vary for different configurations. In each case we include the value to which the *maximum search depth* (MSD) is set (see section 2.4) and the length of the longest path reached during verification (depth). (Notice that in the cases where an error is reported, the depth reached depends on the value to which MSD has been set.) The value given in the 'States' column indicates the number of states stored during the search and 'Memory' denotes the memory (in Mb) required for state storage. A $\surd$ in the 'Result' column indicates that the associated property is satisfied. This means that no error is reported during the verification of properties 1,2 or 4, or that an error (counter-example) is reported during the verification of property 3. Note that in the verification of property 1, a node with more than one neighbour (a non-leaf node) is chosen. This is because otherwise the property is trivially true (as p is always false).

For all verification runs we used a PC with a 1500 MHz Pentium 4 processor and 500Mb of main memory running the Linux operating system (kernel 2.4.17).

Property 1: This property is violated if there exists an infinite path along which p remains true. This is equivalent to there being a path in which a cycle exists, throughout which p is true. An important point here is that when a property of this type (a *liveness* property) is checked in SPIN, it is necessary to apply *weak fairness*. The weak fairness condition ensures that any process that has a transition that remains enabled will eventually execute it (see section 2.4). If, for example, property 1 with $i = 2$ is tested against the configuration of figure 1(a) with no weak fairness, an error will be reported by SPIN (via an acceptance cycle) and a trace in which process 2 reaches the *wait_for_request* (TIP) or *found_partner2* (MTIP) label and simply fails to progress to *child_handshake* (TIP) or *become_parent* (MTIP) (despite process 2 having an enabled transition due to a full channel) will be provided as a counter-example. This sort of scenario is of course of no interest. However, with weak fairness selected, process 2 will be forced to take its enabled transition and no error is reported. When property 1 is tested against the cyclic configuration of figure 1(b) (with weak fairness selected and $i = 2$), an error is reported because process 2 is unable to make a transition from *wait_for_request* (or *found_partner2* in the MTIP case) to a state other than *wait_for_request*.

Property 2: With the use of weak fairness this property is verified for all values of i, for all acyclic configurations. Weak fairness can be avoided by considering a slightly weaker property, in which we limit the search to traces for which each process does eventually arrive at the *finish* state. This avoids the situation in which processes simply fail to progress. Originally we defined this property as:

Property 2(a) *A leader will always be elected before all processes reach the finish state.* That is $(q\mathcal{P}t)$ where q is $\neg(elected == N)$ and t is $((node[proc0]@finish) \wedge (node[proc1]@finish) \wedge \ldots \wedge (node[procN-1]@finish))$. Here we use the *precedes* operator $\mathcal{P}$ where $f\mathcal{P}g = \neg(\neg fUg)$.

This property was verified successfully for all acyclic configurations on 6 nodes. However, we have assumed that once a process reaches the *finish* state it remains there indefinitely. But, this is not the case, after a process has reached the *finish* state it can terminate (and so no longer have a state associated with it). Therefore, the possible scenario in which no leader is elected and all processes reach the *finish* state – *but not at the same time* – could be missed. Thus it is necessary to introduce a new array ($finished$) to record a process passing through the finish state. Once process i reaches the *finish* state the corresponding element in the $finished$ array is set to 1. Hence property 2(a) can be refined thus:

Property 2(b)
$(q\mathcal{P}t)$ where q is $\neg(elected == N)$ and t is $((finished[0] == 1) \wedge (finished[1] == 1) \wedge \ldots \wedge (finished[N-1] == 1))$.

This property was easy to verify and gave us additional confidence that the system (model) behaved as expected.

Property 3: To prove property 3 it was only necessary to show that $neg\langle\rangle r$ is **not** true. That is, it is only necessary to find a single counter-example to show that Property 3 holds. In all cases a counter-example is found very quickly. In table 2 MSD has been set to 10000 (in order to maintain consistency). However a smaller value (greater than 755) could have been chosen which would have resulted in a smaller memory requirement.

Property 4: Although a search of the entire search space was required, no *weak fairness* was necessary (failure to progress did not prevent the property from being satisfied), and property 4 was easily verified.

7.1. Comparison of results for the TIP and MTIP models

Both models satisfy property 1 for all acyclic configurations and all of properties 2, 2(b), 3 and 4. In addition they both incur an error (acceptance cycle) during the verification of property 1 for acyclic configurations. Comparing tables 1 and 2 we see that the values for the MTIP are larger than the corresponding values for the TIP model, but – apart from the verification of property 1 with configuration (b)– of the same order of magnitude. The reason that the values (for depth, Mem and Time) are larger for the MTIP model, is that

Table 1. Results of Verification of the TIP model

Prop	WF?	Config*i*	MSD ($\times 10^4$)	States ($\times 10^5$)	Depth	Mem (Mb)	Time (s)	Result
1	$\surd$	b,2	13	0.2	125646	1.0	2	$\times$
	$\surd$	a, 2	2	6.5	17807	18.8	195	$\surd$
2	$\surd$	a, -	2	8.9	16807	25.9	424	$\surd$
2(b)	$\times$	a, -	2	4.4	15768	12.6	26	$\surd$
3	$\times$	a, 0	1	0.001	755	2.3	0.1	$\surd$
4	$\times$	a, 0	2	4.5	15768	16.7	27	$\surd$

Table 2. Results of Verification of the MTIP model

Prop	WF?	Config*i*	MSD ($\times 10^4$)	States ($\times 10^5$)	Depth	Mem (Mb)	Time (s)	Result
1	$\surd$	b,2	44	3.2	436781	10.2	248	$\times$
	$\surd$	a, 2	4	11.8	38255	33.7	381	$\surd$
2	$\surd$	a, -	4	13.5	39170	38.3	629	$\surd$
2(b)	$\times$	a, -	4	6.8	38144	17.1	40	$\surd$
3	$\times$	a, 0	1	0.002	763	2.0	0.1	$\surd$
4	$\times$	a, 0	4	6.8	38144	16.7	40	$\surd$

in this case each non-leaf process has to return to the *start* state every time that a child has been assigned to it, unlike the TIP model, in which $n-1$ children are assigned simultaniously. However, the similarity of the values for the different models indicates that the behaviour is otherwise essentially the same.

During the verification of property 1 for configuration (b) the values obtained vary more markedly. An acceptance cycle is found far more quickly in the TIP model. This is because, in the TIP model, leaf nodes 0 and 1 will send *be_my_parent* requests to node 2 and then become blocked in the *response* state, and the remaining nodes will only progress as far as the *wait_for_request* state before all becoming blocked. In the MTIP model however, the *be_my_parent* requests will be acknowledged (allowing the leaf processes to terminate) and node 2 will complete 2 full cycles (returning to the *start* state in each case) before becoming blocked at the *wait_for_request* state.

8. Analysis of systems with a large number of processes

Although model checking is useful, to provide a general picture of a system and for finding errors, we are limited to being able to verify properties for a *specific* system and for a *specific* configuration, for a small number of processes. The Tree Identification stage of the FireWire protocol is a good example. We can show, using Promela and Spin, that for all networks $N \leq 6$ nodes a model representing the behaviour of this network, where each individual process behaviour is determined by the tree-identifier protocol, satisfies certain properties. When the number of nodes is increased the memory and time requirements for verification increase, until eventually verification becomes intractable. Ideally, we would like to show that, from the verification of certain properties for all (acyclic) configurations of $N = 6$ processes, say, it is possible to infer the properties for all (acyclic) configurations of N processes.

This is an example of the *parameterised Model Checking problem* which is, in general, undecidable [AK86]. The verification of parameterized networks is therefore often accomplished via theorem proving [KMS00, OSR92, RR01], or by synthesising network invariants [CGJ95, KM89, WL89, BCG89]. Both of these approaches require a large degree of ingenuity.

In some cases it is possible to identify subclasses of parameterised systems for which verification is

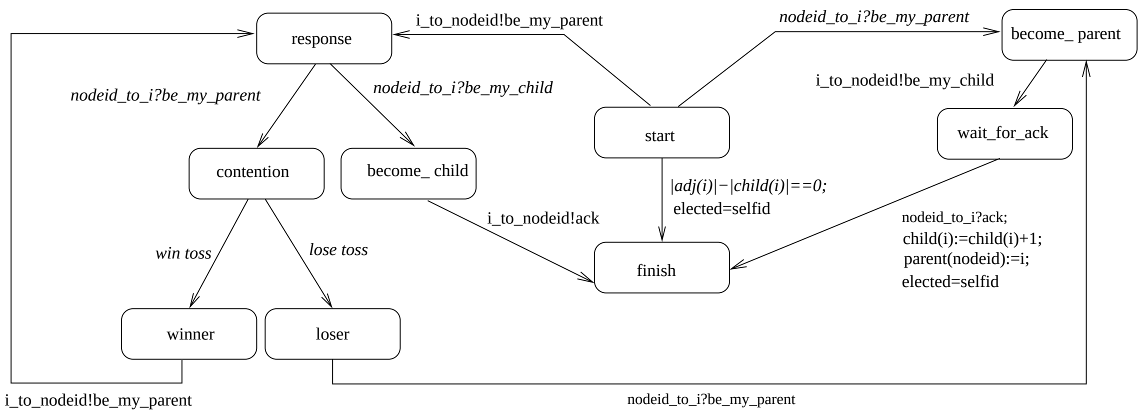

Fig. 5. Finite state automaton representing leaf i behaviour

decidable. Examples of the latter mainly consist of systems of N identical processes communicating within a ring topology [EN95, GS92] or systems consisting of a family of N identical *user* processes together with a *control* process, communicating within a star topology [KMOS94, GS92, ID99]. A more general approach [EK00] considers a general parameterised system consisting of several different classes of processes.

One of the limitations of both the network invariant approach and the subclass approach is that it can only be applied to systems in which each component (contained in the set of size N) is completely independent of the overall structure of the system: adding an extra process (to this set) does not change the semantics of the existing components. A generalisation of data independence is used to verify arbitrary network topologies [CR99, CR00] by lifting results obtained for limited-branching networks to ones with arbitrary branching.

All of these methods fail when applied to asynchronously communicating processes like ours, where processes communicate asynchronously via shared variables. In addition, the type of property that these approaches are used to verify tend to be safety properties. We would like to also prove liveness properties (eg. property 2).

As the MTIP progresses, processes terminate in the sense that they no longer play any part in the protocol. This *degenerative behaviour* of the protocol, means that the approaches described above can not be directly applied, to prove properties of a system of N nodes *for any* N. However, this aspect of the behaviour of the protocol suggests that an alternative approach to induction, exploiting the fact that once a process has terminated, the system will degenerate to a system of N nodes. This would imply that (certain) properties that hold for all systems of N nodes will apply to systems of $N+1$ nodes. Note that this degenerative behaviour does not exhibit so apparently in the TIP as groups of nodes are assigned (essentially) at the same time. (Indeed, this is why we chose to consider the MTIP in the first place.)

In this section we first describe how abstraction can be used to allow us to prove the properties given in section 5 for the MTIP for star topologies for very large (ie. over 100) N. We then show how, with minor modification and under a *degeneration assumption*, we can prove our properties for all star-topologies, i.e. for *any* N.

8.1. The MTIP for a star topology for fixed N

The nodes in a star topology consist of $N-1$ leaf processes together with one non-leaf process. The behaviour of the non-leaf process is given in figure 4. The behaviour of the leaf processes is more restricted, and although this behaviour is contained within figure 4, it can be described more succinctly by the reduced automaton of figure 5. Note that *nodeid* is the *id* of the non-leaf process.

For a star topology (with $N-1$ leaves) a simpler Promela model for the MTIP can be constructed, consisting of one node process and $N-1$ leaf processes running concurrently. Verification of property 2 (“a leader will always be elected”) for a star topology for 9 processes using this simplified Promela model of the MTIP was possible, but slow due to the need to apply weak-fairness (Time= $1176s$, memory for state-storage = $146Mb$). However, with even as few as $N=10$ nodes, even using this simplified Promela model of the MTIP, a complete search was impossible. Despite using a machine with 1.5Gb of main memory (but substantially slower than the PC used for our previous experiments, see section 7) the available memory was eventually exhausted (after 21 hours).

In order to show that a leader is always elected, we are not interested in the internal behaviour of the processes, only the fact that one process eventually reaches the start state with no remaining unassigned neighbours and thus becomes the leader. Similarly, to show that process i can be elected leader, the internal behaviour of all processes other than i can be ignored.

This observation leads us to an abstraction of the MTIP for the star topology for a given N and given property. Each abstraction consists of up to 3 processes (assuming that at most one free variable occurs in the property).

Suppose that we wish to verify a property p with n free variables (where $n \leq 1$), for a system of N processes with a star-topology. Let j denote the *id* of the process that is not a leaf. The abstracted model consists of a node process with $id = j$, a process *Abs* representing the leaf processes with $id \neq j$ and $id \neq k$ for any free variable k occurring in p, together with at most n leaf processes with $id = k$, for free variables k. For example, consider a star topology on 10 processes, where 0 is the *id* of the non-leaf node. If we wish to prove property 2 (which has no free variables), the abstracted model consists of a node process (with $id = 0$) together with a process *Abs* representing the remaining 9 processes. Similarly, to prove property 3 which has one free variable i, if $i \neq 0$, then 3 processes are required, but if $i = 0$ then only 2 processes are required (as process i is not a leaf in this case).

The process *Abs* has states similar to those of the simplified leaf process given in figure 5, except that the *start* and *finish* states are replaced by *all_at_start* and *all_at_finish* respectively and the other states (*response, contention*, etc.) are replaced by states with the prefix "*one_at_*" (namely *one_at_response, one_at_contention* etc.). The "*all_at_start*" state is reached when all of the remaining leaf processes represented by the *Abs* process that remain unassigned are at the *start* state. The "*all_at_finish*" state is reached when all of the leaf processes represented by *Abs* have reached the *finish* state. The states having the "*one_at_*" prefix are reached when at least one of the leaf processes represented by *Abs* have reached the corresponding state given in the leaf process of figure 5. Note that, as *Abs* has $id = N$, the new default value (for the *leader* variable for example) is $N + 1$. Thus, in property 2, q is now defined as $\neg(elected == (N + 1))$.

Apart from the new state labels, the automaton for the *Abs* process is similar to that for the leaf process. However, upon reaching the *one_at_become_child* state, if the number of remaining unassigned leaves within the *Abs* process (denoted by a global variable, *no_leafs_left*) is equal to zero, the process now moves to the *all_at_finish* state. Otherwise, the process returns to the *all_at_start* state.

8.2. Verification Results for the Abstracted MTIP model

Scripts are again used to generate the appropriate model from a (new) template. For a given N and property, a model is generated that consists of the appropriate *node*, *Abs* and (if necessary) *leaf* proctype declarations and associated Promela specification. It is now possible to verify all of the properties given in section 6 for large N quickly and well within our memory allocation. (For example with $N = 100$, during the verification of property 2 the depth reached is only 3690 and the number of (stored) states 3414.) Details of the Promela specification are omitted here due to space restrictions, but an example is contained on our website at [CM].

8.3. Verification of the MTIP for a star topology for any N

However efficient the abstraction of section 8.1 may be, it still has limited use, in that it can only be used to verify properties for a fixed value of N. However, if we let $MA(N)$ denote the abstracted model of size N, we can easily convert $MA(N)$ to a new model M such that $M \supseteq MA(N)$ (M contains all of the behaviour of $MA(N)$) for all values of N. (That is M is an invariant for $MA(N)$ [BCG89, WL89]. The proof that M is an invariant is omitted here, and forms the basis of current work.)

This conversion would involve two minor modifications. Firstly, in the *Abs* process, the *no_leafs_left* variable is originally set to 2. Upon reaching the *one_at_wait_for_ackb* state this variable is non-deterministically assigned a value of 0, 1 or 2. Secondly, the *node* process, instead of checking the value of $|adj(i)| - |children(i)|$, now checks the value of *no_leafs_left* plus the number of unassigned leaves external to *Abs* (0 or 1) and acts accordingly.

This approach could only be used to prove safety properties. (Clearly "a leader is always elected" will no longer be true, as the value of *no_leafs_left* may never equal zero.) However, under the assumption that all processes eventually terminate (which we call the *degeneration assumption*) which would imply that *no_leafs_left* eventually equals zero, all of the properties could be verified.

The implementation of this approach is omitted, for space reasons.

9. Comparison with other approaches contained within the Workshop Proceedings

In this section we compare our approach with some others contained within the *workshop proceedings*. Thanks to the valuable feedback we received at the workshop and from anonymous referees thereafter, our paper has developed considerably since our initial abstract submission. We fully appreciate that, likewise, the final papers corresponding to the abstracts referred to below, may be very different to the workshop version upon which we have made our comparisons. Unlike ours, none of the approaches appear to attempt to generalise results to configurations of N processes, for all N, even for specific subclasses of topologies.

The approach that compares most with our paper is "A Simple Verification of the Tree Identify Protocol with SMV" by Schuppan et al.

Their approach is similar to ours, in that finite-state model checking is used to verify certain properties, and suffers the same limitations as ours, in that (finite state) model checking can only be applied to a "fixed configuration" of processes, and so, even for a fixed number of nodes, each possible configuration must be checked separately. They use a C-preprocessor to create each of their models, whereas we use a Perl script to automatically generate each model and the subsequent model checking run. In SMV each process is described using guards that describe the possible states of the other processes. Therefore, a change to the number (N) of processes, or to the configuration of N processes requires many changes to be made to the (SMV) process description. However, in SPIN, each process is described via a generic *proctype* description, which does not depend on the particular value of N. Thus a change to N or to the topology merely requires (global) connection information to be updated.

The approach of Schuppan et al (mainly) uses synchronous communication. All communication within our model is asynchronous. Also, all of our runs are *non-deterministic*, and so all possible behaviour is checked, whereas their approach is largely deterministic. It is unclear whether the use of the expression "deterministic configuration" (when the force-root flag is fixed) corresponds to the deterministic choice of bits described earlier in the paper, or indeed to deterministic behaviour in general.

Their model is, we believe, closer to the operational view than our own– more attention is paid, for example, to particular time-constraints, and signals. In addition, their paper considers the force-root condition, whereas ours does not.

Another approach which seems similar to ours is that of Verdejo et al who specify and verify the protocol using rewriting logic. They also use state-space exploration to check all possible behaviours of their system. However, in their abstract they do not specify their properties explicitly so it is difficult to make a true comparison. They appear to perform a manual check on the endstates reached by exploration (to check that a unique leader is elected), rather than use the model checker to check a property. This approach would not be applicable to check more complicated properties. It would not, for example, be appropriate for the verification of precedence properties.

10. Evaluation

We have used Promela to model both the tree identify protocol (TIP) and a modified version of the tree identify protocol (MTIP). The reason for considering both versions is that the MTIP lends itself to generalisation (whereas the TIP does not). We have analysed the entire tree identification phase in each case. Our analysis takes the form of verifying that a set of properties hold, using the model checker SPIN.

Our models are easy to modify – it was very straightforward to adapt the TIP model to create the MTIP model for example (or vice versa as, historically, is the case). However, any change to the model requires that all verifications must be repeated.

The original model (of the MTIP) which appears in the workshop proceedings took about 1 week to construct and fully verify. The production of a Perl script enabling one to automatically generate a MTIP model for any number of nodes and any configuration took a further 2 weeks. The TIP model was constructed in a couple of days.

The greatest amount of time was spent in the generalisation of the MTIP. The initial abstracted model took a man-month to produce. This is mainly due to the effort required to understand how the degenerative behaviour of the MTIP could be exploited.

As the framework involved (modelling with Promela) is very close to concurrent programming, an experienced concurrent programmer would quickly grasp the basics required to understand the used specification framework. However, one of the most difficult (and time-consuming) parts of LTL model checking is the specification of logical properties that accurately describe the aspects of the model to be checked. The time that it would take to become proficient in this respect defies quantification.

In order to understand our solution no detailed knowledge of Promela and SPIN is required as all relevant aspects are explained in the text. However, it would take the average programmer a matter of weeks to fully appreciate the subtleties involved.

Not only does our TIP model provide a suitable alternative formalisation to the IEEE standard, but our abstracted MTIP model enables us to investigate the behaviour of any number of processes for a certain subclass of topologies.

11. Conclusions and Further work

We describe how the SPIN model checker can be used to verify a set of essential properties of the Tree Identify protocol (TIP), together with a modified version of the tree identify protocol (MTIP). We illustrate the use of LTL model checking and demonstrate the importance of correctly specifying properties and optimising the code using refinement and reinitialisation. We present full results of our analysis of networks of six nodes and compare the results obtained for the TIP and MTIP models.

We give an abstraction for verifying the MTIP for networks of N processes for a star topology, where N is large and describe how this abstraction can be modified to allow verification of properties for all networks of N nodes having a star topology, for all values of N. Future work involves the extension of this approach to more general network topologies.

References

[AK86] Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986.

[Ara99] H. R. Arabnia, editor. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, volume II, Las Vegas, Nevada, USA, June – July 1999. CSREA Press.

[BCF01] Gérard Berry, Hubert Comon, and Alain Finkel, editors. *Proceedings of the thirteenth International Conference on Computer-aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, Paris, France, July 2001. Springer-Verlag.

[BCG89] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81:13–31, 1989.

[Bos99] Dragon Bosnacki. Partial order reduction in presence of rendez-vous communication with unless and weak fairness. In *[DGLM99]*, pages 40–57, 1999.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In *[LS95]*, pages 395–407, 1995.

[CGM$^+$97] Alessandro Cimatti, Fausto Giunchiglia, Giorgio Mingardi, Dario Romano, Fernando Torielli, and Paolo Traverso. Model checking safety critical software with SPIN: an application to a railway interlocking system. In *[Lan97]*, pages 5–17, 1997.

[Cho74] Yaacov Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[CM] M. Calder and A. Miller. Veriscope publications website: `http://www.dcs.gla.ac.uk/research/veriscope/publications.html`.

[CM01a] M. Calder and A. Miller. Feature interaction analysis using the model-checker SPIN. Technical Report TR2001-91, University of Glasgow, Department of Computing Science, 2001.

[CM01b] M. Calder and A. Miller. Using SPIN for feature interaction analysis - a case study. In *[Dwy01]*, pages 143–162, 2001.

[CR99] S.J. Creese and A.W. Roscoe. Formal verification of arbitrary network topologies. In *[Ara99]*, 1999.

[CR00] S.J. Creese and A.W. Roscoe. Data independent induction over structured networks. In *[PDP00]*, 2000.

[DGLM99] D. Dams, R. Gerth, S. Leue, and M. Massink, editors. *Theoretical and Practical Aspects of SPIN Model Checking:*

Proceedings of the 5th and 6th International Spin Workshops, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[DH99] Matthew B. Dwyer and John Hatcliff. Slicing software for model construction. In Olivier Danvy, editor, *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, pages 105–118, San Antonio, Texas, January 1999. University of Aarhus. Technical report BRICS-NS-99-1.

[Dwy01] M.B. Dwyer, editor. *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, Toronto, Canada, May 2001. Springer-Verlag.

[EK00] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *[McA00]*, pages 236–254, 2000.

[EN95] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *[POP95]*, pages 85–94, 1995.

[EN98] E. A. Emerson and Kedar S. Namjoshi. On model checking for nondeterministic infinite state systems. In *13th IEEE Symposium on Logic in Computer Science*, pages 70–80, 1998.

[GHP96] J.-Ch. Gregoire, G.J. Holzmann, and D. Peled, editors. *Proceedings of the Second Workshop on the SPIN verification System*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Rutgers University, New Brunswick, August 1996. American Mathematical Society.

[God96] P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th international Conference on Protocol Specification Testing and Verification (PSTV '95)*, pages 3–18. Chapman & Hall, Warsaw, Poland, 1995.

[GS92] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, July 1992.

[Hol90] Gerard J. Holzmann. Algorithms for automated protocol verification. Technical Report 69, AT & T, January/February 1990.

[Hol93] Gerard J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25:981–1017, 1993.

[Hol97a] Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Hol97b] Gerard J. Holzmann. State compression in Spin: Recursive indexing and compression training runs. In *[Lan97]*, 1997.

[Hol98] Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, November 1998.

[Hol99] Gerard J. Holzmann. The engineering of a model checker: The gnu i-protocol case study revisited. In *[DGLM99]*, pages 232–244, 1999.

[HP99] G.J. Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *International Journal on Software Tools for Technology Transfer*, 2(3):270–278, November 1999.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[HS99] G.J. Holzmann and Margaret H. Smith. Software model checking - extracting verification models from source code. In *[WCG99]*, volume 156, pages 481–497, 1999.

[ID96] C.Norris Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.

[ID99] C. Norris Ip and David L. Dill. Verifying systems with replicated components in Murϕ. *Formal Methods in System Design*, 14:273–310, 1999.

[IEE95] IEEE 1394-1995. *IEEE Standard for a High Performance Serial Bus Std 1394-1995.* Institute of Electrical and Electronic Engineers, August 1995.

[IEE00] IEEE 1394a-2000. *IEEE Standard for a High Performance Serial Bus (Supplement) Std 1394a-2000.* Institute of Electrical and Electronic Engineers, 2000.

[Kap92] Deepak Kapur, editor. *Automated Deduction - Proceedings of the 11th International Conference on Automated Deduction (CADE 1992)*, volume 607 of *Lecture Notes in Computer Science*, Saratoga Springs, NY, USA, June 1992. Springer-Verlag.

[KM89] R. P. Kurshan and K.L. McMillan. A structural induction theorem for processes. In *Proceedings of the eighth Annual ACM Symposium on Principles of Distrubuted Computing*, pages 239–247. ACM Press, 1989.

[KMOS94] Robert P. Kurshan, M. Merritt, A. Orda, and S.R. Sachs. A structural linearization principle for processes. *Formal Methods in System Design*, 5(3):227–244, December 1994.

[KMS00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[Lan97] R. Langerak, editor. *Proceedings of the Third SPIN Workshop (SPIN'97)*, Twente University, The Netherlands, April 1997.

[LS95] Insup Lee and Scott A. Smolka, editors. *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR '95)*, volume 962 of *Lecture Notes in Computer Science*, Philadelphia, PA., August 1995. Springer-Verlag.

[LS97] Siegfried Löffler and Ahmed Serhrouchni. Creating a validated implementation of the steam boiler control. In *[Lan97]*, 1997.

[McA00] David A. McAllester, editor. *Automated Deduction - Proceedings of the 17th International Conference on Automated Deduction (CADE 2000)*, volume 1831 of *Lecture Notes in Computer Science*, Pittsburgh, PA, USA, June 2000. Springer-Verlag.

[MP90] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. Technical Report STAN-CS-90-1321, Stanford University, June 1990.

[MRS01] S. Maharaj, J. Romijn, and C. Shankland, editors. *Proceedings of the International Workshop on Application of Formal Methods to IEEE 1394 Standard*, Berlin, Germany, March 2001.

[MS00] Kamel M. and Leue S. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):394–409, 2000. Special section on SPIN.

[MT00] Lynette I. Millett and Tim Teitelbaum. Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. *International Journal on Software Tools for Technology Transfer*, 2(4):343–349, 2000.

[OSR92] S. Owre, N. Shankar, and J. Rushby. PVS: A prototype verification system. In *[Kap92]*, pages 748–752, 1992.

[PDP00] *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, volume II, Las Vegas, Nevada, USA, June 2000. CSREA Press.

[Pel96a] D. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design*, 8:39–64, 1996.

[Pel96b] Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *[PPH96]*, pages 233–257, 1996.

[POP95] *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages (POPL '95)*, San Francisco, California, January 1995. ACM Press.

[PPH96] Doron A. Peled, Vaughan R. Pratt, and Gerard J. Holzmann, editors. *Proceedings of the DIMACS Workshop on Partial-Order Methods in Verification (POMIV '96)*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[RR01] A. Roychoudhury and I. V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *[BCF01]*, pages 25–37, Paris, France, July 2001. Springer-Verlag.

[Sif89] J. Sifakis, editor. *Proceedings of the International Workshop in Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, Grenoble, France, June 1989. Springer-Verlag.

[VB96] Willem Visser and Howard Barringer. Memory efficient state storage in Spin. In *[GHP96]*, pages 185–203, 1996.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings, Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society, June 1986.

[WCG99] Jianping Wu, Samuel Chanson, and Quiang Gao, editors. *Formal Methods for Protocol Engineering and Distributed Systems: Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '99)*, volume 156 of *International Federation For Information Processing*, Beijing, China, October 1999. Kluwer.

[WL89] Pierre Wolper and Vinciane Lovinfosse. Properties of large sets of processes with network invariants (extended abstract). In *[Sif89]*, pages 68–80, 1989.

Appendix A

We include here the Promela model generated for the MTIP for 6 node processes, with the configuration of figure 1(a), to verify property 2. Notice that the never-claim associated with property 2 occurs at the end of the code.

```
/*Modified Leader Election model with 6 processes generated from template*/
/*With property 2: A leader will always be elected */

mtype = {nullmessage,be_my_parent,be_my_child,ack};

#define N 6  /* no. of nodes */
byte adj[N]; byte child[N]; byte toss[N]=0; byte elected=N;
typedef array { byte to[N] }; array connect[N];
chan null = [1] of {mtype};

/*define the channels, need to change per configuration*/

chan zerotwo=[1] of {mtype}; chan twozero=[1] of {mtype}; chan onetwo=[1] of {mtype}; chan twoone=[1] of {mtype};
chan twothree=[1] of {mtype}; chan threetwo=[1] of {mtype}; chan threefive=[1] of {mtype}; chan fivethree=[1] of {mtype};
chan fourfive=[1] of {mtype}; chan fivefour=[1] of {mtype};


inline converter(id1,id2,chanin,chanout)
/* takes a pair of ids and finds the corresponding in and out channels */
{if
 :: (id1==0)-> assert((id2==2)); chanin=twozero;chanout=zerotwo
 :: (id1==1)-> assert((id2==2)); chanin=twoone;chanout=onetwo
 :: (id1==2)-> assert((id2==0)||(id2==1)||(id2==3));
    if
    :: (id2==0)->chanin=zerotwo;chanout=twozero
    :: (id2==1)->chanin=onetwo;chanout=twoone
    :: (id2==3)->chanin=threetwo;chanout=twothree
    fi
 :: (id1==3)-> assert((id2==2)||(id2==5));
    if
    :: (id2==2)->chanin=twothree;chanout=threetwo
    :: (id2==5)->chanin=fivethree;chanout=threefive
    fi
 :: (id1==4)-> assert((id2==5)); chanin=fivefour;chanout=fourfive
 :: (id1==5)-> assert((id2==3)||(id2==4));
    if
    :: (id2==3)->chanin=threefive;chanout=fivethree
    :: (id2==4)->chanin=fourfive;chanout=fivefour
    fi
 fi}

proctype node(byte selfid)
{mtype message=nullmessage;

 byte counter=N; byte partnerid=N;
 chan self_in=null; chan self_out=null;

 start:
  atomic{
  assert((counter==N)&&(message==nullmessage)&&(partnerid==N)&&(self_in==null)&&(self_out==null));
  if
  ::(adj[selfid]-child[selfid]==0)->elected=selfid; goto finish
  ::(adj[selfid]-child[selfid]==1)->counter=0; goto parent_request
  ::else->counter=0;goto wait_for_request
  fi};

 parent_request:
  atomic{
  assert((partnerid==N)&&(message==nullmessage)&&(self_in==null)&&(self_out==null)&&(counter<N));
  if
  ::((counter!=selfid)&&(connect[selfid].to[counter]==1))->
    partnerid=counter; counter=N; goto found_partner1
  ::else->counter++;goto parent_request
  fi};

 found_partner1:
  atomic{assert(partnerid!=selfid&&(self_in==null)&&(self_out==null)&&(message==nullmessage));
  converter(selfid,partnerid,self_in,self_out);
  if
  ::self_in?message->assert(message==be_my_parent); message=nullmessage; goto become_parent
  ::assert(empty(self_out));self_out!be_my_parent;goto response
  fi};

 response:
  atomic{
  full(self_in);assert((message==nullmessage)&&(counter==N));self_in?message;
  if
  ::message==be_my_child->message=nullmessage; partnerid=N; goto become_child
  ::message==be_my_parent->message=nullmessage; goto contention
  fi};

 become_child:
  atomic{
  empty(self_out); self_out!ack; assert((message==nullmessage)&&(counter==N)&&(partnerid==N));
  self_in=null;self_out=null;goto finish};

 contention:
  atomic{
  assert((message==nullmessage)&&(counter==N)&&(partnerid!=N));
  if
  ::selfid<partnerid->counter=selfid
  ::else->counter=partnerid
  fi;
  if
  ::(toss[counter]==0)->toss[counter]++; counter=N;goto winner /*win toss*/
  ::else->assert(toss[counter]==1);toss[counter]=0; counter=N;goto loser /*lose toss*/
  fi};

 winner:
  atomic{
```

```
  empty(self_out);self_out!be_my_parent;goto response};

 loser:
  atomic{
  self_in?message;assert(message==be_my_parent); message=nullmessage; goto become_parent};

 wait_for_request:
  atomic{
  assert((partnerid==N)&&(self_in==null)&&(self_out==null)&&(message==nu llmessage)&&(counter<N));
  if
  ::(connect[selfid].to[counter]==0)->counter++;
    if
    :: counter==N->counter=0
    :: else->skip
    fi;
    goto wait_for_request
  :: (connect[selfid].to[counter]==1)->assert(counter!=selfid);goto found_partner2
  fi};

 found_partner2:
  atomic{
  converter(selfid,counter,self_in,self_out);
  if
  :: full(self_in)->partnerid=counter;  counter=N; self_in?message; assert(message==be_my_parent);
     message=nullmessage; goto become_parent
  :: empty(self_in)->counter++;
  if
  :: counter==N->counter=0
  :: else->skip
  fi;
  self_in=null;self_out=null; goto wait_for_request
  fi};

 become_parent:
  atomic{
  empty(self_out); assert((message==nullmessage)&&(counter==N)&&(partnerid!=N));
  self_out!be_my_child; goto wait_for_ack};

 wait_for_ack:
  atomic{
  full(self_in); self_in?message; assert(message==ack); message=nullmessage;
  connect[selfid].to[partnerid]=0; connect[partnerid].to[selfid]=0;
  self_in=null;self_out=null;partnerid=N; child[selfid]++; goto start};

}
/*end of node process*/

#define q (elected!=N)

init
{atomic{
/*set connect.to array */
 connect[0].to[2]=1; connect[2].to[0]=1; connect[1].to[2]=1; connect[2].to[1]=1;
 connect[2].to[3]=1; connect[3].to[2]=1; connect[3].to[5]=1; connect[5].to[3]=1;
 connect[4].to[5]=1; connect[5].to[4]=1;
/*set neighbours array */
 adj[0]=1; adj[1]=1; adj[2]=3; adj[3]=2; adj[4]=1; adj[5]=2;
/*run processes*/
run node(0); run node(1); run node(2); run node(3); run node(4); run node(5);}}

/* Formula As Typed: []  <>  q
* The Never Claim Below Corresponds to The Negated Formula !([]  <>  q)
* (formalizing violations of the original)*/

never {    /* !([]  <>  q) */
T0_init:
if
:: (! ((q))) -> goto accept_S2
:: (1) -> goto T0_init
fi;
accept_S2:
if
:: (! ((q))) -> goto accept_S2
fi;}
```