SMC: A Symmetry based Model Checker for Verification of Safety and Liveness Properties*

A. P. Sistla[†] V. Gyuris[‡] E. A. Emerson[§]

April 13, 1999

Abstract

The paper presents the SMC system for checking safety and liveness properties of concurrent programs under different fairness assumptions. It is based on explicit state enumeration; it combats the state explosion problem by utilizing symmetries of the input concurrent program usually present in the form of identical processes. It reduces the number of explored states by identifying those states that are equivalent under the symmetries of the system; this is called *process symmetry.* It reduces the number of edges explored from each state by exploiting the symmetry of a single state, called *state symmetry.* SMC works in an *on-the-fly* manner, it constructs the reduced state graph as and when it is needed. This method facilitates early termination, speeds up model-checking and reduces memory requirements.

We employed SMC to check the correctness of, among other standard examples, the Link Layer part of the IEEE Standard 1394 "Firewire" high-speed serial bus protocol. SMC found some deadlocks in the protocol. SMC was also used to check certain liveness properties. A report on the case study is given as part of the paper.

1 Introduction

Recently there has been much interest in the academic as well as the industrial community in developing systems for automatically verifying finite state concurrent programs. Such systems have been successfully used to verify real-life protocols and hardware circuits. Their wider applicability is hampered by the state explosion problem, i.e., the enormous increase in the size of the state

*The work of Sistla and Gyuris is partly supported by the NSF grants CCR-9623229, CCR-9212183, CCR-9633536. The work of Emerson was supported by the NSF grant CCR-9804736

[†]Electrical Engineering and Computer Science Department, The University of Illinois at Chicago, USA

[‡]Mathematics, Statistics and Computer Science Department, The University of Illinois at Chicago, USA

[§]Compute Science Department, University of Texas at Austin, Austin, Texas.

space of real systems. In this paper we describe the system SMC (Symmetry based Model Checker) that exploits the symmetry in the concurrent program to contain the state explosion problem.

SMC can be used for model-checking safety and liveness properties of concurrent programs. Typically, inputs to SMC include the description of a concurrent program and a Buchi automaton on infinite strings. The automaton specifies all the incorrect computations (i.e. it represents the complement of the correctness property; note that such automata are more powerful than Linear Temporal Logic (LTL) and every property specified in LTL can be systematically translated in to a Buchi automaton). SMC checks if there exists a *fair* computation of the concurrent program that is accepted by the Buchi automaton, i.e., it checks if the program exhibits an incorrect fair computation (under the counter-example option, one such computation will be output). The salient features of SMC are the following:

- It checks correctness/incorrectness under two different notions of fairness — *weak fairness* and *strong fairness* — as specified by the user;
- It *exploits symmetries* in the concurrent program, usually present in the form of identical processes, to speed up model-checking as well as to reduce memory requirements. This is done in two different ways. Firstly, it reduces the number of explored states by exploiting inter-state symmetry. This is achieved by identifying those states that are equivalent under the symmetries of the system and exploring the graph from only one or a few number of states belonging to each such equivalence class. We call this as the *process symmetry.* It reduces the number of edges explored from each state by exploiting intra-state symmetry, i.e. the symmetry of each state, which is called *state symmetry.* This is achieved by partitioning the set of processes in to groups so that all processes in a group are, roughly speaking, in the same local state, and by exploring the transitions of only one process in each group (called the leader process).
- It is an *on-the-fly model-checker*, i.e., it constructs the reduced state graph as and when it is needed. SMC is based on the theory developed in [10, 11, 14]. Additionally, SMC is equipped with an explicit *deadlock detection* mode that detects existence of reachable deadlock states; this mode is invoked if the input does not include the automaton.

During the model-checking process, SMC needs to periodically check if two given global states are equivalent under the symmetry of the system. It employs some novel randomized algorithms for checking the equivalence of states. Further more, it employs elegant methods for performing the on-the-fly model-checking. The salient features of SMC are its use of state symmetry in addition to the process symmetry, and its novel on-the-fly algorithm that checks for correctness under fairness while exploiting symmetry.

We employed SMC to check the correctness of many standard examples. We found SMC to perform well on the systems that exhibit high levels of symmetry.

For example, we were able to handle simple resource controller examples with up to 200 processes which would have been impossible to handle without exploiting symmetry. Here we found that process symmetry together with state symmetry substantially reduces the computational time and memory usage.

As a case study, we applied SMC for checking the correctness of the link layer part of the IEEE Standard 1394 "Firewire" high-speed serial bus protocol. We modeled the link layer part of the protocol in our input language. We discovered two potential problems in the protocol. The first one is a deadlock which was also detected earlier by other researchers [31]. The second one is regarding the following liveness property: a site that sends a data message should eventually receive an acknowledgment from the receiving site. We found that this property is not satisfied, by our model, even under strong fairness. After careful examination of the counter example given by SMC, we found that satisfaction of the above property requires transition fairness in the physical layer process. Using this fairness we were able to check that the protocol satisfies the above liveness property. (We did not model the physical layer part of the protocol. The physical layer was represented by a single process that modeled its interactions with link layer and the actual transmission of data. Due to this reason, we do not know if this bug in the model constitutes a bug in the over all protocol. Our results only show a potentail problem which needs to be further examined by a detailed analysis of the physical layer implementation. Our results also show that the actual implementation of the physical layer should guarantee transition fairness in order to satisfy the liveness property). There has been earlier work [31, 28, 24] on analyzing the link layer part of the Firewire protocol (see section 7). However, ours is the first work that analyzed the liveness properties of the protocol, in addition to safety properties, using the model-checking approach.

The paper is organized as follows. Section 2 describes related work and compares SMC with other systems. Section 3 describes the main features of the system. Section 4 describes the syntax and semantics of the input language. Sections 5 and 6 present the details of the different components of the system with emphasis on the implementation. Section 7 presents experimental results including the case study. Summary and conclusions are given in section 8. The interested reader should consult [14] for further details of the algorithm and for theoretical background.

2 Related Work

Many model-checking systems have been developed, over the years, for the analysis and verification of concurrent systems. Some of the prominent ones are SMV [30], Murphi [19], SYMM [22, 5], SPIN [18], COSPAN [27], HyTech [1], KRONOS [16] (a more elaborate list can be found in [27]). Many of these systems ([4, 13, 18, 27]) use on-the-fly techniques to avoid storing the complete state space in the main memory. Some of these systems (e.g. COSPAN) perform on-the-fly model-checking and handle weak fairness.

To the best of our knowledge, ours is the first approach, and SMC is the first implemented system, that performs on-the-fly model checking under fairness (both strong and weak fairness) for the full range of temporal properties and that exploits symmetry.

The above mentioned systems employ a variety of techniques for combating the state explosion problem. Among these, Murphi and SYMM are the systems that are closest to ours. Both these systems employ symmetry based techniques for combating the state explosion problem. Murphi is an explicit state, on-the-fly model-checker. It primarily checks for invariant properties (i.e. properties specified by CTL formulas of the form $AG(p)$ where p is a state predicate); it does not handle general temporal properties. The SYMM system combines symmetry with BDDs for checking the full range of properties that can be specified in the logic CTL. However, both Murphi and SYMM do not handle fairness properties, i.e. they do not check for correctness under fairness. As a consequence they can be mainly used for safety properties only. On the other hand, SMC can check for correctness under weak fairness as well as strong fairness. Thus, SMC provides the additional functionality of being able to check the full range of liveness properties.

An additional advantage of SMC, compared to Murphi and SYMM, is the the exploitation of *state symmetry*. Roughly speaking, the state symmetry of any global state partitions the processes in to groups such that all the processes in the same group are in the same local state. While exploring the reachable state space, SMC only executes the enabled transitions of a single process from each group. On the other hand, both Murphi and SYMM execute all enabled transitions from a global state. The use of state symmetry reduces over all time, as the number of invocations to the equivalence checker routine are reduced.

As indicated earlier, the input to SMC consists of specification of a concurrent program and an incorrectness specification given as an automaton. The language for specifying the concurrent program is similar to the input language of Murphi. We employ modules to denote a class of identical processes; this is similar to scalar sets of Murphi. While SMC is process oriented, Murphi is data oriented. All the variables of the concurrent program are assumed to be of integer type (thus it is possible to specify an infinite state program in this language; in this case, either SMC finds an execution accepted by the automaton and terminates, or it may not halt).

Symmetry has also been employed in COSPAN [27] and was also employed in protocol verification [3, 26] where data symmetry was employed. [27] shows how to reason about temporal properties under certain types of fairness like assumptions; however, fairness is not handled in full generality and the approach is limited to cyclic symmetry.

Symmetry based techniques for checking the full range of temporal properties were first given in [5, 10]. Although the initial methods of [5, 10] could only handle a limited set of liveness properties, a more generalized approach for checking liveness properties under various notions of fairness has been proposed in [11]. This method, however, does not facilitate early termination, it supplies an answer only after the construction of all the required data structures is complete.

Symmetry based techniques have been employed in the Petri-net community [21, 23]; here, the research has been primarily on checking for existence of deadlocks.

In the literature many other techniques have been proposed for combating the state explosion problem of model checking. One such technique is the method based on Symbolic Model-checking as implemented in SMV [30]. This method represents groups of states symbolically as OBDDs (Ordered Binary Decision Diagrams) and performs model-checking using the OBDDs. This system has been quite successful in hardware verification.

Another technique is based on partial order reductions [13, 17, 33]. This method is based on explicit state enumeration; here the size of the explored state space is reduced by only considering certain inter-leavings of executions of concurrent processes. This method has been implemented in the SPIN system [18].

3 Main Features and Overview

This section describes the main features of SMC.

3.1 State space exploration with fairness

SMC is based on explicit state enumeration. Such methods typically have three basic phases: (i) they construct the global state graph; (ii) they compute the product of the global state graph and the incorrectness automaton; (iii) and finally they perform a search for a strongly connected subgraph of the product graph that satisfy certain properties. The last phase in the above method is based on the observation that infinite incorrect computations correspond to strongly connected subgraphs of the product graph containing a final automaton state; such subgraphs are called final strongly connected subgraphs.

Fairness conditions, in general, are necessary and critical for checking liveness properties of concurrent systems. There are two standard notions of fairness properties, identified to be important, in the literature [12] — *weak fairness* and *strong fairness*. A computation is said to be *weakly fair* if every process that is continuously enabled from a particular point in time is eventually executed after that point. A computation is said to be *strongly fair* if every process that is enabled infinitely often in the computation is executed infinitely often. As mentioned earlier, SMC handles both weak and strong fairness.

Fairness of an infinite computation can be formulated as a property of the corresponding strongly connected subgraph (called *fair strongly connected subgraphs*). A strongly connected subgraph C is *weakly fair* if the following condition is satisfied for every process p: there exists a state in C in which p is disabled or C contains an edge caused by the execution of a transition of process p. Similarly, a strongly connected subgraph C is *strongly fair* if the following condition is satisfied for every process p: if C contains a state in which

p is enabled then it also contains an edge that is caused by the execution of a transition of p. It has been shown that the input program has a weak fair (or strong fair) incorrect computation iff the product graph contains a final and weakly fair (resp. strongly fair) strongly connected subgraph that is reachable from the initial state.

COSPAN [27] was one of the first systems that used the above method for checking correctness under weak fairness. Efficient algorithms that check for existence of strongly fair strongly connected subgraphs were first given in [9].

The above method does not allow us to handle even a moderately sized concurrent program due to the *exponential explosion* in the size of the global state graph with the number of processes.

3.2 Compression by symmetry

One possible way to reduce the state space to be explored is to utilize the symmetries of the concurrent program. The symmetry existing in the concurrent program induces an equivalence relation on the global states. By keeping only a single representative state from each equivalence class we can compress the state space to a smaller structure called *Annotated Quotient Structure* (AQS). We call this techniques as *process symmetry*. An edge of the AQS linking two representative states s and t corresponds to edges of the original state space between two states that belong to the equivalence classes of s and t respectively. AQS edges are annotated with the index of the process whose execution created the edge, and it is also labeled with a permutation. This permutation on the label indicates how process indices need to be permuted in order to get the state in the original state graph as one traverses along the edge. Symmetry based techniques for temporal logic model-checking were introduced [19, 5, 10]; these works mostly used quotient structures with out annotations. AQS was introduced in [11] and was used for checking correctness under different fairness conditions.

In addition to compression based on equivalence of states, the implemented model checker exploits the internal symmetry of a single global state, called *state symmetry*, to further speed up the model-checking. The state symmetry defines a partition on the set of processes; roughly speaking, all processes in the same local state are in the same group of the partition. For each group, only edges corresponding to a single process, called the leader process, are constructed and stored. Some of the edges that are not stored are reconstructed as and when needed from the stored edges using the internal symmetry structure of the AQS state. A preliminary version of state symmetry was introduced in [10]. This was expanded and was integrated in to the model-checking algorithm in [14].

SMC supports three types of symmetry options.

1. This option has no symmetry. This can be used if there is no symmetry in the system or the number of identical processes is too small; use of other symmetry options in these cases may slightly slow down because of the overhead involved.

2. Under this option only process symmetry is used. AQS is constructed by identifying equivalent states and no state symmetry is invoked.

3. This option employs both process and state symmetries. In systems with much symmetry and many processes, this option saves both computation time as well memory. However in some cases, because of the extra overhead, this option may be slower than the previous option.

3.3 On-the-fly construction

In SMC we broke the sequential line of the above mentioned general algorithm. SMC performs the three phases (AQS construction, product graph construction, search for strongly connected subgraphs) in an integrated manner. An on-the-fly algorithm, that exploits symmetry and checks for correctness under different fairness properties, was presented in [14]. SMC employs this algorithm.

SMC supports the following three on-the-fly options for the construction of the AQS.

1. In the first option the entire AQS is constructed in advance. Its advantage lies in the possibility of reusing the constructed AQS in checking multiple correctness properties. Its main disadvantage is that it constructs parts of the AQS that may not be needed, causing increase in time and memory usage.

2. In the second option, the nodes and the edges of AQS are constructed when they are needed for the first time in the product graph construction. The constructed edges are stored so that they need not be reconstructed when needed by the product graph generator at a later time. This option constructs only the parts of the AQS that are needed and, hence, tends to be faster than the first option.

3. Finally, in the most space efficient case, the nodes and edges are constructed, as in the previous case, in an on-the-fly manner but the edges are not stored. The edges from a given AQS node are constructed as and when needed (in the worst case, as many times as the number of product graph nodes that refer to the AQS node). This option uses less space than the previous one but it may take more time.

We adopt the standard depth-first search algorithm, due to Tarjan [32, 2], for computing the strongly connected components, for checking the existence of a reachable fair strongly connected subgraph of the product graph. Recall that this is a recursive algorithm that also employs an additional stack where the nodes of the partially explored strongly connected components are stored. We modify and integrate this algorithm with the generation of the product graph (i.e. the algorithm is executed on-the-fly as the nodes of the product graph are generated). In order to check that the strongly connected subgraph is fair (weakly or strongly fair), we maintain a *state similarity* partition with each vertex on the stack. This partition divides the processes in to groups so that all processes in the same group satisfy the same fairness property (If one of the processes in the group satisfies the fairness property then we conclude that all the processes in the group satisfy the fairness property). A more detailed presentation of this algorithm is given in section 6. A formal description of the algorithm is given in [14].

The on-the-fly method has two main advantages when compared with the sequential one. The first advantage is that it permits early termination. When the already explored part of the product graph contains a fair strongly connected subgraph the system detects it and terminates immediately saving the effort of constructing the rest of the AQS and the product graph. The second advantage is that the on-the-fly method reduces the amount of memory used. When either option two or three (from the above three options) is used , then only the needed portion of the AQS is constructed and no edges of the product graph are stored. When option three is used then even the edges of the AQS are not stored. We found that when option three is used the memory usage gets substantially reduced, while the execution time increases substantially.

For the examples we considered, the state symmetry utilization and the on-the-fly option (more specifically, option two) makes SMC 75–90% faster and 70–85% more space efficient than its earlier version which did not use these advanced techniques (see [11]).

4 Input Language

4.1 Concurrent Program Specification

Syntax

SMC assumes that the input program has the following architecture. Processes are divided into modules. All processes in a module are identical up to renaming; that is, any permutation mapping processes in a module to processes in the same module is a symmetry of the system. This assumption helps automatic symmetry detection in the AQS construction, since the symmetry group of the program itself is very easy to represent. We can certainly simulate a system of some distinct, non-identical processes by declaring *each* of them to be a module containing a single process.

The BNF notation describing the syntax of the input language is given in the appendix. An input concurrent program consists of a sequence of commands separated by semicolon. Each command is a module declaration, a variable declaration, an index variable declaration, a constant declaration or a module specification. A module declaration declares a module and specifies the number of processes in that module. A module specification specifies the transitions in a generic process of that module. All processes in a module are identical up to renaming. An index variable declaration, specified by the production rule for the non-terminal `<ind-var>`, declares an index variable. Each index variable ranges over the processes of a single module that is declared to be the type of the index variable. The only operations that can be performed on index variables are equality and inequality testing. In particular, index variables can not be updated. A variable declaration defines a set of program variables. Each such declaration specifies a name and a list of index variables. Each such declaration defines a single program variable for each combination of values to the corresponding index variables; thus, it declares a multi-dimensional array of variables. A variable declaration without any list of index variables declares a single variable. A variable declaration also specifies the initial values of the declared variables. Program variables are the variables through which processes communicate. All program variables are of integer type.

All variables and their initial values need to be declared explicitly, index variables may be declared implicitly. A module specification starts with the declaration of a single index variable, called the *indicator index variable*, that is followed by a set of transition schemas. The indicator index variable shows the process to which the instance of the transition schema belongs to. Each transition schema is given by a condition part and an action part. The condition part is a boolean expression over atomic conditions. An atomic condition is a program variable compared with an expression or is simply a variable. The indices of program variables have to be indexed variables, i.e. they can not be constants (constants as indices will destroy the symmetry).

The action part is a set of assignments. In the condition part as well as in the assignment part we are allowed to use quantification over process indices of a module such as `ALL(`c` of client: `<*expression or assignment*>`)`. Every occurrence of c in the above clause is said to be a bound occurrence. If the above clause appears in the condition part, then it is replaced by a conjunction of terms obtained by substituting the different possible values for the index variable. An existential quantifier over index variables can appear in the condition part; this is expanded in to a disjunction. Universal quantification over index variables can appear in the action part; existential quantifiers can not appear in the action part. An action of the form `ALL(`c` of client: lc[c] = 0)` is expanded in to a set of actions corresponding to different values of c. Note that use of quantifiers over index variables does not break symmetry. Any occurrence of an index variable, which is not a bound occurrence, is called a free occurrence. We assume that all occurrences of an index variable in a transition schema are free or all its occurrences are bound. The free index variables of a transition schema are all those index variables, other than the indicator index variable, that appear free in the schema.

The processes in a module are defined from the module specification as follows. Let p be the indicator variable of the module specification and let n be the number of processes in the module (specified in the module declaration). For each value of p such that $0 \leq p < n$, the transition in process p of the module are defined as follows. Let T be a transition schema in the module specification and let $x_1, x_2, ..., x_n$ be all the free index variables in T. Note that these index variables range over the process indices of the modules that define their types. Let m_i be the number of processes in the module that defines the type of x_i. An evaluation ρ over the above free index variables is a mapping that, for each $i = 1, ..., n$, associates a value $\rho(x_i)$ such that $0 \leq \rho(x_i) < m_i$.

For each evaluation ρ, process p has a transition obtained from the schema T by substituting $\rho(x_i)$ for each of the free index variable x_i.

Comments are specified by // as in C language. The following example program defines client-server scheme in which clients access resources controlled by the servers.

Consider the following example.

```
// declaration of modules
Module server          = 2;
Module client          = 3;

// declaration of program variables
busy[server]           = 0;
request[server, client]  = 0;
reply[server, client]  = 0;
lc[client]             = 0;

//index variable declaration
s of server;
c of client;

// server specification starts
s: busy[s] == 0 && request[s,c] == 0 -> reply[s,c] =1,
                                        busy[s] = 1;

// client specification starts
c: {
    lc[c] == 0                     -> lc[c] = 1, request[s,c] = 1;
    lc[c] == 1 && reply[s,c] == 1 -> lc[c] = 2, request[s,c] =0;
    lc[c] == 2 && reply[s,c] == 1 -> lc[c] = 0, busy[s] =0,
                                     reply[s,c]=0;
   }
```

The above example declares two modules called `server` and `client` having two and three processes, respectively. It also declares two sets of variables `reqst[`s`,`c`]`, `reply[s,c]`, for each $0 \leq s < 2$ and $0 \leq c < 3$. For each server s, it declares the variable `busy[`s`]`. For each client c, it declares the location counter variable `lc[`c`]`. All these variables are initialized to zero. It also declares two index variables s, c ranging over server and client processes respectively.

The server module has a single transition schema (note that s is used as the indicator index variable). For each value of $s = 0, 1$, the server process s has three transitions obtained by substituting the three values $0, 1, 2$ for the client index variable c. The client module has three transition schemas (note c is the index variable). For each value of $c = 0, 1, 2$, the client process c has six transitions obtained by substituting $s = 0, 1$; note that s appears in all the transition schemas of this module.

Semantics

The semantics of such a concurrent program is similar to the transition systems of Manna and Pnueli [29] or Unity language of Chandy and Mishra [8]. A global state of such a program is a mapping that associates appropriate values to all the program variables. An initial global state is one in which all the global variables are assigned their initial values. A transition t of some process p is said to be enabled in a global state s if the condition part of the transition is satisfied in the state s. A process p is said to be enabled in state s if some transition of p is enabled in s. A process or a transition is said to be disabled in a state if it is not enabled in that state. A state s is called a *dead-lock* state if none of the transitions of the processes are enabled in s. Let s, s' be two global states. We say that s' is obtained from s by executing a transition t of some process p, if t is enabled in s and s' is obtained by executing the action part of t (i.e. corresponding to each action of the form $x = e$, the expression e is evaluated in the state s and the resulting value is assigned to x). We say that s' is obtained from s by executing process p if s' is obtained by executing some transition of p.

A computation $\sigma = s_0, ..., s_i, ...$ is an infinite sequence of states satisfying the following properties:

- s_0 in an initial state.
- For each $i \geq 0$, s_{i+1} is obtained from s_i by executing an enabled transition of some process, or s_i is a dead-lock state and $s_{i+1} = s_i$.

The computation $\sigma = s_0, ..., s_i, ...$ is *weakly fair* if for every process p, there are infinite number of values of i such that p is disabled in state s_i or s_{i+1} is obtained from s_i by executing p; i.e. either p is disabled infinitely often or is executed infinitely often. The computation σ is said to be *strongly fair*, if for every process p the following condition holds: either p is disabled continuously from certain point (i.e. there exists an i such that for all $j \geq i$, p is disabled in s_j), or p is executed infinitely often in σ (i.e. there are infinite number of values of i such that s_{i+1} is obtained from s_i by executing process p).

Comparison to Murphi

Our input language is similar to that of Murphi (actually, we independently designed the language and later found the similarity with Murphi). Our modules correspond to the scalar sets of Murphi. Our language is process oriented, i.e. each module is considered a collection of processes. Murphi is data oriented and does not have the concept of process. Murphi allows more complex control structures such as loop constructs (*while* statement), conditional statements. Where as our model is similar to the transition systems of [29]. It is well known that any complex control structure (see [29]) can be translated in to our scheme by using variables representing location counters.

Murphi allows complex data structures such as records etc. Having such complex structures is certainly convenient, but they do not add any additional expressive power. We can certainly translate such complex data structures, with some inconvenience, in to our system by declaring the fields of a record as independent variables. It is fairly straightforward to extend our system to support such data structures.

Murphi allows program variables that can store process identities (note that our index variables range over process ids, they can not be updated and they are not program variables). Such variables are useful in sending messages containing process identities. Although we do not directly support such program variables, we can use the following approach to represent such variables. In order to simulate a variable x (of Murphi) that contains a process id belonging to a module M, we declare a variable $X[M]$. Note that such a declaration defines an array of size m where m is the number of processes in the module M. At any point in time, at most one variable in the array has value 1, while all other variables have value zero. Wherever x is set to some value p, there $X[p]$ is set to one and all other variables in the array X are set to zero. Using this approach, we will certainly increase the size of a global state by a factor of at most n, where n is the total number of processes. However, this does not change the number of reachable states in the global state graph.

4.2 Automata Specification

The automata specification language consists of a declaration part and a list of automata transitions. The declaration part specifies the initial and final states and possibly the list of tracked processes. The tracked processes are those processes that are monitored by the automaton. The BNF for specifying an automaton is given in the appendix.

An automata transition is of the form `condition` $\rightarrow$ `state1 : state2`. Here `condition` is a boolean expression. There is an important difference between the condition part of an automaton transition and that of a transition schema in the module specification of the input program. The condition part in the automaton transition may contain constants in the index list of a variable; this is not permitted in the condition part of transition schema of a module. The production rules corresponding to the non-terminal `<condition>` are not given as it is rather complex. The non-terminal `<number>` generates a sequence of decimal digits, while `<name>` specifies an identifier.

If the list of tracked processes is not explicitly specified by the automaton specification, it is automatically generated; in this case, processes directly referenced in the automaton (denoted by constants in the index list of a variable) are taken as the tracked processes. Tracked processes require special attention in the product graph construction. One of the conditions for the equivalence of two product states is that their tracked process components should be equal.

The following is an example automaton specification.

```
InitialState M0;
FinalState M1;
\\ transitions

True           -> M0:M0;

EXISTS(i:EXISTS(j: i!=j & lc[i]==2 & lc[i]==2)) -> M0:M1;

True -> M1:M1;
```

The above automaton has two states $M0, M1$ and has three transitions. The condition part of the second transition is satisfied if there exists two client processes i and j such that their location counter values are both equal to two; for the client-server program given earlier, this indicates that both the processes are accessing the resource at the same time. For a system, with one server (i.e. only one resource), this denotes violation of mutual exclusion. The type of an index variable is deduced from the context. Also, $EXISTS$ is an existential quantifier on index variables. The above automaton has no tracked processes.

The following automaton has one tracked process which is process 0 of the client module.

```
InitialState M0;
FinalState M1;
\\ transitions

True           -> M0:M0;
lc[0]==1       -> M0:M1;
lc[0]==1       -> M1:M1;
```

5 Construction of AQS

This section is devoted to describing the necessary background and the implementation details of the AQS construction routine.

5.1 Generation of the AQS

A permutation on processes that maps every process in a module to a process in the same module is called a symmetry of the system. The set of all symmetries forms a group. For any state s and symmetry π, let $\pi(s)$ denote the state such that the value of the variable $\texttt{v}[i_0, \ldots, i_{n-1}]$ in state $\pi(s)$ equals the value of the variable $\texttt{v}[\pi^{-1}(i_0), \ldots, \pi^{-1}(i_{n-1})]$ in state s. Let π be a symmetry. For any two states s, t if state t can be reached from s by executing a transition of process i then $\pi(t)$ can be reached from $\pi(s)$ by executing a transition of the process $\pi(i)$. We say that two states s, t are equivalent if there exists a symmetry π such that $t = \pi(s)$. It is well known that this binary relation is an equivalence relation among the states [19, 5, 10]. In our AQS construction, we identify all such equivalent states and represent them by a representative state belonging to the equivalence class. In our construction, the representative of an equivalence class, of states, is the first generated state from that class.

Now we describe state symmetry. Let s be a state. We say that process i and j are *equivalent in state* s if $\pi_{i,j}(s) = s$ where $\pi_{i,j}$ is the permutation that interchanges i and j but fixes all other processes[1]. This relation is an equivalence relation among the processes. This equivalence relation induces a partition on the set of processes. We call this partition as *the state symmetry partition of* s. For each equivalence class in the state symmetry partition, we pick a representative process belong to that class; this representative process is called the *leader* of that class. During the construction of the AQS, we will only consider the leader processes and execute their transitions. Edges corresponding to the execution of non-leader processes can be generated from those of leader processes [14]. This approach saves AQS construction time, and it also saves space as we dont have to store the edges corresponding to non-leader processes.

Our AQS construction algorithm works as follows. During the operation of the algorithm, the set of already generated states are classified in to those that are "yet to be explored" and those that have already been explored. Exploring a state involves generating all the edges leaving that state. In the beginning, the explored set is empty and the yet to be explored set consists of the initial states. The algorithm repeatedly does the following. It chooses an yet to be explored state s and processes it as follows. First the state symmetry partition of s is computed and the leader processes of each class of the partition are nominated. Then each transition of each of the leader processes is executed in state s. Suppose s' is the resulting state after execution of such a transition. Then the algorithm invokes a function that checks if there exists an already constructed state that is equivalent to s'. If such a state t exists, then a pointer to t together with a permutation π, such that $\pi(t) = s$, is returned. In this case, the AQS generator simply introduces an edge from s to t and labels the edge with π and the index of the process that caused that transition. If no such state exists then s' is stored as a new yet to be explored state. After this, state s is put in the set of explored states.

Edges corresponding to the transitions of non-leader processes need not stored. They can be generated from the edges caused by the leader processes as follows. Let s be an AQS state and let process i be the leader of process j. This means that $\pi_{i,j}(s) = s$. Assume that execution of a transition of j creates an edge from s to some state t'. This means that execution of a transition of i causes an edge from $\pi_{i,j}(s) = s$ to $\pi_{i,j}(t')$. The states t' and $\pi_{i,j}(t')$ are equivalent and have a common representative. Let t be the common representative and ρ be the symmetry such that $\rho(t) = \pi_{i,j}(t')$. We only store the AQS edge

[1] This is a slightly more restrictive notion of state symmetry than the one introduced in [10]. However, it is easy to implement, i.e. there exists efficient algorithms for detecting equivalence of two processes in a state.

$s \xrightarrow{\rho,i} t$ only and not the AQS edge $s \xrightarrow{(\pi_{i,j}\rho),j} t$ (note that since $\pi_{i,j}$ is its own inverse, $\pi_{i,j}(\rho(t)) = \pi_{i,j}(\pi_{i,j}(t')) = t'$). The later edge can be computed from the former as and when needed (it will be needed in very few cases.)

When deadlock option is invoked, we only store the states of the AQS but not its edges. In the AQS construction two technical problems deserve our attention. These are the efficient storing of the constructed states and the equivalence test.

5.2 Hashing

In order to search for the existence of an equivalent state efficiently, a hash table is employed. Whenever we need to search for a state equivalent to s', we compute an index i. This entry in the hash table points to a linked list of states that are hashed to the same location. This linked list is searched. For each state t on the list, it is checked if t is equivalent to s' under some permutation π which is a symmetry of the system. If such a π exists then a pointer to t together with π are returned.

Choosing an appropriate hash function is quite critical here. Firstly, the hash function has to be consistent with the symmetry, that is, all equivalent states should be hashed to the same location. On the other hand, it is desired that inequivalent states should be hashed to different locations.

We used the following method for computing the hash function. For a given state, we first compute an integer called the checksum. Let x be the checksum value. We compute the hash value to be $x \bmod b$ where b is the hash table size. The checksum of a state is stored with the state and is used in a quick pretest as explained in the next subsection. It is computed as follows. First, we add all the values of variables belonging to an array together (i.e. we add values of all variables that have the same name but correspond to different index values). Let $n_0, n_1, ..., n_{k-1}$ be the resulting integers (note that k is equal to the number of program variable declarations; the ordering of these numbers is same as the order in which the variables are declared in the program). We compute checksum to be equal to $\Sigma_{0 \leq i < k} n_i (11)^i$. It is easy to see that the above hash function is consistent with the symmetry[2].

We also use another hash table where we store all the permutations that label AQS edges. When a new AQS edge , with permutation π, is created , we search for π in the hash table; if it is not present then we insert it in the hash table. We store a pointer to the permutation in the AQS edge.

5.3 Checking equivalence of states

Before checking the equivalence of two states, we check if their checksums are equal. If they are not equal then the two states are not equivalent and the

[2]In an earlier version, we also used a multiplication hash function where we increment all the variable values and multiply them to obtain a number x, and use $x \bmod b$ as the hash index. This method has advantage that the hash index is independent of the checksum. However, it increased the overall time by about 10% as we need to perform extra computation.

procedure exits with a negative answer. Otherwise, the equivalence test is carried out. Experimental results have shown that this quick pretest saves lot of computational time (the pretest caught more than 99% of the cases when the two tested states are inequivalent).

Checking equivalence of states even for the above model of processes is a hard problem, as it is equivalent to the graph isomorphism problem [5]. Our equivalence checking algorithm is an approximation algorithm that runs in quadratic time. Whenever it indicates that two given states are equivalent then they are equivalent; however, the algorithm may err reporting that two states are not equivalent in some cases when they actually are. But this error yields that two equivalent states are stored as distinct ones instead of storing only one of them. In this way the compressed AQS will not be optimal, it will contain redundancies. This will effect the speed of the computation but certainly not the correctness. So the one sided error in the performance of the equivalence test algorithm will not pose a danger to the algorithm.

The equivalence detection algorithm has two phases. The first phase is based on the *naive refinement process* used in approximation algorithms for graph isomorphism problem [15]. A similar approach has also been used in checking equivalence of states in Murphi.

5.3.1 First Phase of the Algorithm

Below we describe the first phase of the algorithm. Suppose that the equivalence of states s and t is to be tested. Assume that processes k and l belong to the same module, say M_0; further assume that there is a variable `v` indexed by the modules M_0 and M_1 (i.e. recall that `v` is a two dimensional whose first and second dimensions range over process indices in M_0 and M_1, respectively) and x is a possible value for `v[i,j]`. Clearly, if the size of the set $\{i \in M_1 \ : \ \mathtt{v}[k,i] = x \ \textit{in state } s\}$ is not equal to that of $\{i \in M_1 \ : \ \mathtt{v}[l,i] = x \ \textit{in state } t\}$ then there is no symmetry π that maps process k to l and such that $\pi(s) = t$. This property leads to the notion of a sequence of successively complex *signatures*. The 0^{th} signature $\textsc{Sig}_0(k,s)$ of a process k in state s is simply the name of the module to which k belongs. We can define further refined signatures inductively based on the previous ones. The $(n+1)^{\text{th}}$ signature $\textsc{Sig}_{n+1}(k,s)$ of a process k in state s is the unordered multi-set (set with possibly repeating elements) of all tuples of the form $\langle x, \mathtt{v}, i, \textsc{Sig}_n(p_0,s), \ldots, \textsc{Sig}_n(p_{c-1},s)\rangle$ satisfying the following properties: `v` is the name of a variable shared between modules $M_0, \ldots, M_{c-1}$; $p_0, \ldots, p_{c-1}$ are processes belonging to modules $M_0, \ldots, M_{c-1}$ respectively; $p_i = k$, and the value of $\mathtt{v}[p_0, \ldots, p_{c-1}]$ in state s is x. Note that $\textsc{Sig}_n$ is used as a component in $\textsc{Sig}_{n+1}$. It can be proven by induction that these complex signatures have the useful property that $\textsc{Sig}_n(k,s) = \textsc{Sig}_n(l,t)$ holds whenever s and t are equivalent under some permutation that maps k to l.

A formal description of *phase 1* of the algorithm is given in the appendix. In this phase, the algorithms computes the n^{th} signature function for the two tested states during the n^{th} iteration. If, at any time, the number of processes with some signature S in the two states differ then the algorithm exits reporting that the states are not equivalent. If, on the other hand, after step $n+1$ the new signatures are not finer[3] than the ones in step n then the algorithm jumps to *Phase 2*. *Phase 1* is called *the naive refinement process*, for further details consult [15].

5.3.2 Second Phase of the Algorithm

In *Phase 2* a permutation π of the processes is constructed such that $\textsc{Sig}_n(k,s)$ equals $\textsc{Sig}_n(\pi(k),t)$. We do this by mapping a process i to process j if the signature of i in state s equals to that of j in t. Of course, there could be multiple processes with the same signature. In that case we perform a random matching. Finally, the algorithm checks if the permutation is indeed a valid equivalence. If most of the processes have a unique signature then the number of possible permutations is limited and the random matching has high probability of finding an equivalence. Furthermore, by repeating the random matching many times we can increase the probability of success. If running time is not a concern, we can test all possible matchings. That would result a 100% reliable algorithm that runs (in very symmetric cases) in exponential time.

There are other variants of the above described symmetry detection algorithm that, in general, err in more cases but runs considerably faster. For example, we can limit the number of iterations performed during phase 1 of the algorithm as that is the most time consuming.

Notice that the above algorithm is a polynomial time bounded randomized algorithm. It may indicate two states to be inequivalent even though they are equivalent.

For the following special case, however, we have an efficient complete algorithm that runs within polynomial time bounds. Many resource controller algorithms fall in to this category.

Assume that modules in the system are divided into two major classes: *server* modules and *client* modules. Each server module contains exactly one process while clients may contain multiple processes. Suppose that no variable is shared between two process that belong to client type modules. That means that the index list of a variable can contain at most one client, all other processes are of server type. It can be proven that after only one iteration step in *Phase 1* any random matching will provide a valid equivalence if there is one at all. Hence, in this special case our simplified algorithm always correctly indicates if two states are equivalent.

5.3.3 Comparison to Murphi's Approach

Since we are using an approximation algorithm for checking equivalence, we may have multiple states in the AQS from the same equivalence class. It is interesting to compare our methods for generating the AQS with the methods of Murphi. Murphi works as follows. Whenever a transition of the program is executed, the generated state is converted in to a representative in its equivalence class

[3]That is, $\textsc{Sig}_{n+1}(k,s) = \textsc{Sig}_{n+1}(l,s) \Rightarrow \textsc{Sig}_n(k,s) = \textsc{Sig}_n(l,s)$ for all k and l.

and this representative is hashed in to the hash table and is searched for. If the representative is not found then it is inserted in to the hash table, otherwise it is ignored. There can be multiple representatives in an equivalence class. Finding a representative of a state is a hard problem and Murphi uses some approximation algorithms for it. Thus Murphi executes this expensive conversion algorithm each time a program is executed. In our case, when we generate a state after executing a transition, we hash it and search the hash table for an equivalent state using the approximation algorithm; we check for equivalence only when the pretest is successful, i.e. the checksums match. We believe that in some cases our method is better and in some other cases Murphi's method is better.

There is another difference. In our system we do not execute all enabled transitions in a state, we only execute enabled transitions of the leader processes when the state symmetry option is invoked. On the other hand, in Murphi all enabled transitions in a state are executed. Thus the over all number of times the above procedure is invoked may be much less in our system than in Murphi especially when state symmetry is effective. In these cases, the over all time may be less in our system.

6 Exploration of the Product Graph

A formal description of the Product Graph and the theory behind its construction and the operation of the algorithm can be found in [14]. Here we describe these intuitively from the point of implementation.

6.1 Definition of the Graph

The Product Graph (PG) is the product of the AQS and the automaton. In detail, a state in PG is of the form (s, q, l) where s is an AQS state, q is a state of the automaton and l is the list of local tracked processes. We explain the *local tracked processes* by the following example. Suppose that the automaton states a property about process 1, say, process 1 will never gets into a deadlock situation. As we traverse along the edges in the AQS following a possible execution of the program we may encounter a permutation $\pi_{1,2}$ on an edge indicating that from the next state t onwards process 1 and 2 should be interchanged to get a valid execution. We can achieve the effect of this permutation by continuing the traversal of AQS with the modification that, from state t we check the property as if it were stated for process 2. In other words, we changed the tracked process to 2 *locally*. Continuing the traversal of the AQS we may visit t again after passing through an edge with label $\pi_{2,3}$; from that point onwards, the local tracked will be process 3.

The edges of PG are defined as follows. We have an edge labeled, with the pair (i, π), from (s, q, l) to (s', q', l') if the following conditions are satisfied: $s \xrightarrow{i,\pi} s'$ is an edge of the AQS; the automaton makes a transition from q to q' on the input s concentrating on the local tracked processes l; l' is obtained from

l by the application of permutation π. The edges of PG are generated on-the-fly and are not stored.

We also define an equivalence relation on the states of PG as follows. States (s, q, l) and (s', q', l') are equivalent if $s = s'$, $q = q'$ and each process in list l is equivalent in state s to the process in the same position in list l'. We also define the state symmetry of a PG state. Processes i and j are equivalent in (s, q, l) if $\pi_{i,j}(s) = s$ and $\pi_{i,j}$ fixes all local tracked processes. In other words, for distinct i and j, they are equivalent if neither of them is a local tracked process and i is equivalent to j in the AQS state s. Now it is obvious that the *state symmetry partition* of a PG state can be easily computed from that of the underlying AQS state: we just have to declare each of the local tracked processes as an individual class. As in the AQS case, representative processes, called *leaders*, are chosen from each class.

As indicated in section 3, with each PG state we define a *state similarity partition* which is a partition on the set of processes. Suppose that we find a cycle in PG, that is, a path from a state (s, q, l) to itself. Let π be the product of all the permutations that label the edges of the cycle taken in the order of traversal along the cycle. This cycle denotes a computation of the concurrent program from state s to state $\pi(s)$ and a run of the automaton, on the execution of process l in the computation, that takes it from state q back to itself. Notice that the local state of process i in $\pi(s)$ is same as the local state of process $\pi^{-1}(i)$ in s. Let $k > 0$ be an integer such that π^k is the identity permutation. Repetition of the above cycle corresponds to a computation that starts from state s and ends in the same state and a run of the automaton from state q back to itself. It is not difficult to see that in the above computation processes i and $\pi^{-1}(i)$ go through the same local states (although at different times). Thus both these processes satisfy the same fairness condition. With this motivation, for each PG state $u = (s, q, l)$, we define an equivalence relation on processes: we say that i and j are equivalent in u if there exists a cycle from u back to itself such that the product of permutations on the cycle is some permutation π that maps j to i, i.e. $\pi(j) = i$. The partition defined by this equivalence relation is the state similarity partition of u. This similarity partition helps in finding fair computations. The state similarity partitions of all states in a strongly connected component are identical. They will be computed incrementally as the product graph is constructed. For a formal definition and proofs see [14].

6.2 Finding Fair Strongly Connected Components

Now we can give the details of the PG construction and the procedure for finding fair strongly connected components. The product graph is constructed and simultaneously a strongly connected component search is performed using depth first search. Our algorithm is a modification of the algorithm for strongly connected components, due to Tarjan, given in [32, 2]. Recall that in Tarjan's algorithm, during the depth first search, a separate stack (other than the recursion stack) is used to store the nodes in the partially explored strongly connected components. Each time we explore a new node u, we do the following. We invoke the AQS construction routine for the underlying AQS state. The state similarity partition of u is initialized to the state symmetry partition. Then we process all the transitions that were caused by a leader process. In [14] it has been proven that it is enough to consider transitions of only the leader processes.

During the processing of an edge, if the state, the edge leads to, is equivalent to an already constructed state and that state is still in the stack then the information on the edge label is used to update the state similarity partition of u. This is done because that edge can be used to construct a cycle in PG: it leads to a node from where we can find a path back to the node u. If the transition leads to new node then that node is first fully explored and, if it is still in the stack (this means this node is in the same strongly connected component as u), its state symmetry partition is merged with that of u.

We maintain information on the processes that are known to be fair in the partially explored strongly connected component. We also maintain a flag that indicates if a final automaton state has been encountered in the active strongly connected component (i.e. the partially explored component). Both of these information are updated after an edge of the PG was processed. If we find that all processes are similar to some fair process and the explored active strongly connected component of PG contains a final automaton state then we terminate immediately with "YES" answer. This procedure gives us early termination.

If we reach a maximal strongly connected component we perform different actions in the cases of strong and weak fairness. If weak fairness is checked then the whole component is marked off stack and the command returns to the process that invoked search starting from the first node to have been visited in the strongly connected component. In the case of strong fairness, we first identify those *unfair* processes that are not strongly fair in the component. It means that each such unfair process is enabled in some node of the component but the execution of any transition of that process leads out of the component. As a next step we mark those nodes of the component off stack in which an unfair process is enabled. If all states are marked then we exit the routine. If there are states left then they are copied to a separate queue and treated as new and unvisited ones in the further part of the algorithm. States in the queue are dequeued one after the other and, if they are still not marked off stack, then the above described recursive exploration algorithm is performed on them. A full formal description of the algorithm is given in [14]; this is an adaptation of the algorithm of [9] to the case when we are considering symmetries.

7 Experiments and Case Study

We have tested SMC on various standard examples such as the *Resource Controller* example, *Readers Writers Problem* etc., including a case study involving the IEEE "Firewire" protocol [20]. In all these cases we found symmetry based reduction to be very effective. Whenever the system exhibits much symmetry and the number of processes is large then the third symmetry option (where state symmetry is also used) performs much better than the second symmetry option (where only process symmetry is used); subsection 7.2 contains experimental results that demonstrate this. First, in the following sub-section, we describe our case study.

7.1 Case Study

We used SMC to check the correctness of certain aspects of the IEEE Standard 1394 "Firewire" high-speed serial bus protocol [20] ("Standard" from here on). We concentrated on the link layer part of the protocol as suggested by Roy Koymans (Philips Research, Eindhoven, The Netherlands) suggested[4]. We have modeled the link layer part in detail. However, we did not model the physiacl layer part of the protocol. Since the physical layer part is needed in testing the link layer, we represented the physical layer by a single process that models the following two aspects: (1) the interface of the physical layer with the link layer (i.e. the communication with the link layer part of each site); (2) the actual transmission of the data on the bus.

Our analysis found a deadlock in the link layer part. This deadlock was due to an unspecified operation in a particular state. We corrected this and found no other deadlocks. This deadlock was also discovered earlier by Sighireanu and Mateescu [31] by modeling the protocol in LOTOS and employing the tool box CADP.

We have also checked if the protocol satisfies different liveness properties. We found that an important liveness property, that the sender of an asynchronous message should eventually receive an acknowledgement from the recipient, is not satisfied even under strong fairness. Since we did not model the physical layer part of the protocol, we do not know if this constitutes a bug of the over all protocol. Detrmining this requires detailed modeling of the physical layer which is beyond the scope of this paper. After examining the counter example generated by SMC, we checked the above property, again using SMC, under a stricter notion of fairness called transition fairness (which is explained later). We found that liveness property is satisfied under this fairness. Our conclusion is that the actual implementation of the physical layer should guarantee this transition fairness in order for the liveness property to be satisfied.

There have been other works on analyzing the Firewire protocol. Both Luttik [28], and Kuhn, Hooman and de Roever [24] reported another deadlock in the protocol. While the work of [28] employed muCRL, that of [24] employed PVS; both these systems are based on theorem proving. This deadlock is not present in our model because of the way we modeled the protocol.

To the best of our knowledge, ours is the first attempt on analyzing liveness properties of this protocol using modelchecking based approach.

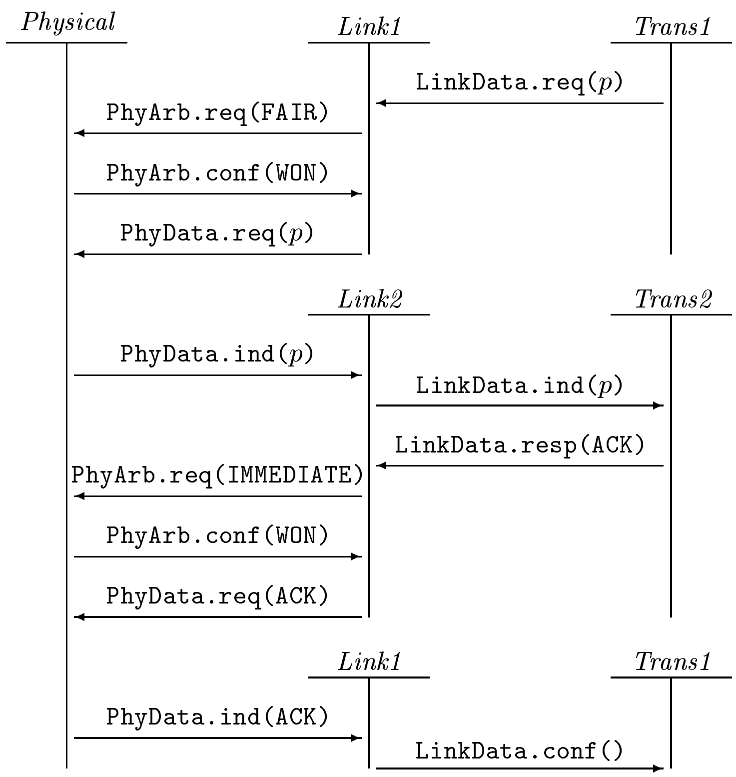

Figure 1: Illustration of a complete communication cycle

Description and Modeling

The simplified description of the asynchronous data transfer mode of the protocol is the following. The link layer serves as an intermediary between the Transaction Layer and the physical layer. The Transaction Layer communicates with the applications, while the physical layer handles the physical transfer of the data. The different layers communicate through *actions* or *services*. When the Transaction Layer wants to send a message p to some other client, it indicates it by executing the action `LinkData.req(`p`)`. The link layer requests arbitration for the bus by executing the action `PhyArb.req(FAIR)`, this is granted by the physical layer through the action `PhyArb.conf(WON)`. The Link Layer then forwards the packet p by the action `PhyData.req(`p`)` to the physical layer. The addressee is notified by the action `PhyData.ind(`p`)` of the incoming packet at the destination site. At this site the message is forwarded by the Link Layer to the Transaction Layer by the execution of the action `LinkData.ind(`p`)`. The Transaction Layer acknowledges the message by `LinkData.resp()` and the ac-

[4]in personal communication

knowledgment should be forwarded back to the sender via the bus. This involves an arbitration with `IMMEDIATE` priority. The action `LinkData.conf()` on the sender side finishes the communication. A schematic illustration of these actions is given in Figure 1.

There are other features of the protocol such as sending a broadcast message, sending a packet through immediate response, error handling, refusal of an arbitration request etc., but we refrain from describing them here. For a complete description of the protocol, the interested reader should directly consult [20].

We employed two major abstractions in the modeling process. Firstly, we considered *time* only implicitly. No clock ticks were modeled, rather we introduced a nondeterministic `TIMEOUT` transition wherever the original description indicated that certain time constraints may be exceeded. As a second abstraction, we assumed that a packet is an abstract entity: various parameters of a packet such as data beginning and end indication, checksum, packet identifier and the data itself were not modeled explicitly.

One of the first conceptual problems in the modeling was translating the notion of an *action* to our shared variable formalism. The standard does not indicate any explicit semantics for the actions. We have used the synchronous semantics which is similar to the CSP semantics. In this semantics, both parties, say the link layer and the physical layer, should execute the action simultaneously (the standard does not specify the semantics; this semantics was also used in [31] and is considered one of the accepted semantics).

There are two ways to implement the synchronized binary communication in our system. The first one is where we model every such possible synchronization as a single transition. If we take this approach then we will have to include this synchronizing transition as part of only one of the two processes (SMC does not permit a transition to belong to multiple processes). This would mean that SMC would consider the execution of this transition as execution of only one process, but not both. As a consequence some fair computations will be considered as being unfair, and hence checking for correctness under fairness will not be sound in general. However, this method is sound for checking correctness of safety properties which do not need fairness. The above approach has the following additional disadvantage. If one of the processes executes the action at n places in its specification and the other at m places then we will have to introduce nm synchronizing transitions in our model corresponding to the different combinations. This proliferation of transitions may not increase the size of the state space but can considerably slow down the construction of AQS.

The second approach is to split the synchronization transition in to two transitions, each one executed by one process, and to serialize the execution of these transitions. The first one to be executed is called the *igniting* transition and the second is called the *handling* transition. In our model we have taken this approach. After execution of an igniting transition all but the corresponding handling transition are disabled in the partner process by setting a special local variable called *pending*. This approach can certainly increase the number of reachable states. This was not a problem since the number of sites considered was at most four.

The link layer and the transaction layer at each site is combined in to another process. These processes at different sites are identical up to renaming their ids. We call these process as the link layer processes.

We modeled the physical layer as a single process. This process, called as the physical layer process, has two different types of transitions. The first type of transitions are the ones that are used to model the binary synchronized communication between the physical layer and the link layer processes. We call these as *one-one* transitions. The second type are those that model the transmission of the data on the physical bus. Whenever a data or an acknowledgment message needs to be transmitted, one of these transitions puts this message in the local buffers of all the link layer processes; these transitions are enabled only when the link layer processes are in states in which they are ready to receive these messages. We call these as the *one-all* transitions.

Thus, we have two modules. The first module contains the link layer processes and the second module contains the physical layer process.

Checking for Deadlocks

After setting the appropriate environment, we checked the Link Layer for deadlocks. We found one deadlock scenario: the one that has already been reported in [31]. It is based on the unintended feature of the link layer that it cannot handle the `LinkData.resp(NO_OP)` given by the Transaction Layer after a broadcast message. The statistics for this case are shown in Figure 2. Each entry in this table has three numbers corresponding to the three symmetry options. The first one corresponds to the case when no symmetry is used, the second one to the case when process symmetry is used but no state symmetry, the third one to the case when both process and state symmetry are used. A * entry indicates that the system ran out of memory, a _ entry indicates that it is the same value as the number just left of it.

Since we are checking only for deadlocks, only AQS states are stored and no edges are stored and no product graph is constructed. The on-the-fly option is ignored.

Note that the number of AQS states is the number generated until a deadlock was found, not the total number of reachable states. We see that use of symmetry has substantially decreased the computation time as well as the memory. For example, when there are four sites then we could not complete the run without symmetry. The numbers for the case when we did not use state symmetry (i.e. used only process symmetry) and the case when we used state symmetry (with process symmetry) are close; as to be expected the number of AQS states and memory are the same; there is slight reduction in computation time when we used state symmetry.

After performing the necessary corrections, SMC showed that the Link Layer is deadlock free. The table below shows the statistics regarding the verifications of the link layer in a network of various number of processes. We used Sun Ultra2 platform with SunOS 5.5.1 operating system. Statistics on our experiments are given in Figure 3. Relative performance of the three different symmetry options

No. of sites	2	3	4	5
AQS states	1670/849/_	23923/4347/_	*/17818/_	*/64710/_
Comp. time (sec)	1/1/1	31/8/8	*/63/60	*/393/344
Memory (kbyte)	293/149/_	5741/1043/_	*/5416/_	*/23818/_

Figure 2: Deadlock case

is same as before. However note that the numbers in figure 3 are higher than the corresponding numbers in figure 2 since SMC had to generate all the reachable AQS states to ensure that there are no deadlocks.

Number of processes	2	3	4
AQS states	2952/1483/_	84752/14575/_	*/89708/_
Computation time (sec)	2/1/1	121/31/31	*/342/337
Memory used (kbyte)	519/261/_	20340/3498/_	*/27271/_

Figure 3: No Deadlock case

Checking Liveness Properties

We checked the following importance liveness property. Whenever the link layer at a site sends a data message, does it eventually receive an acknowledgment? Clearly, we need to use fairness for checking this liveness property. We checked this property under both weak and strong fairness properties. We found that it is not satisfied even under strong fairness. After we looked at the counter example generated by SMC, we found the following scenario. The link layer at one site sends a message and the receiving station sends an acknowledgment. However, before the acknowledgment can be delivered, another station jumps in and tries to arbitrate for the bus and it gets denied continuously. The physical layer spends its time in denying the request and does not get to deliver the acknowledgment.

Here both the symmetry option and the on-the-fly option are important. We used all three symmetry options — NS (No Symmetry), PS (only Process Symmetry), SS (Process and State symmetries). We only used two on-the-fly options: N — no on the fly option, Y — the second on the fly option where the AQS edges are generated and stored when first needed. We did not use the third on-the-fly option where the AQS edges are never stored and are generated as and when needed; this turned out to be too slow. We got the following statistics (shown in figure 4) when we tried with three sites. Each column correspond to a combination of symmetry and on-the-fly options. For eaxmple, a column with heading (PS,Y) corresponds to the case when only process symmetry is used and the on-the-fly option is also used. From the numbers, it is easy to see that states symmetry with the on-the-fly option gives the best performance.

After examining the counter example, we realized that the above problem can be corrected if we further strengthen the strong fairness as follows (which

Options	NS, N	NS, Y	PS, N	PS, Y	SS, N	SS, Y
AQS states	151,190	22,912	25,894	8,639	25,894	6,400
Comp. time (sec)	323	31	59	19	60	13
Memory (kbyte)	43,273	7,023	7,689	2,830	7,690	2,092

Figure 4: Liveness Checking under Strong Fairness

we call as *transition fairness*). For each of the two types of transitions of the physical layer process, i.e. the one-all and the one-one types, if any of the transitions of that type is enabled infinitely often then at least one transition of that type should be executed infinitely often. We can not directly specify such fairness in SMC. Instead, we modeled it by splitting the physical layer process in to two processes each one containing only one of the above two types of transitions. These two processes are put in to different modules. Thus the new system has three modules two containing each of above two processes and another containing the link layer processes. Using SMC with strong fairness, we checked that the above system satisfies the previous liveness property (it is not difficult to see that strong fair computations of the new system of processes are exactly the computations, of the earlier system of processes, that satisfy transition fairness). This shows that actual implementation of the physical layer should guarantee transition fairness in order for the liveness property to be satisfied.

We got the following statistics, given in figure 5, for the new system of processes with three sites. It is to be noted that the numbers in 5 are grater than the corresponding number in figure 4. This is because the system has to generate the whole state space to ensure that no incorrect computations are present. As usual, we get a much better performance using symmetry than when we don't use symmetry. Here state symmetry does not make the performance much better since we only have three sites. When on-the-fly option is used the time is slightly more than when it is not used; this is because of the extra over head and because there is no early termination.

Options	NS, N	NS, Y	PS, N	PS, Y	SS, N	SS, Y
AQS states	151,190	151,190	25,894	25,894	25,894	25,894
Comp. time (sec)	261	274	78	81	81	82
Memory (kbyte)	54,011	54,011	12,978	12,978	12,860	12,860

Figure 5: Liveness Checking under Transition Fairness

7.2 Other Experimental results

As indicated earlier, the third symmetry option, where we use state symmetry together with process symmetry, performs much better than when we only use process symmetry, in those cases when the number of processes is high. We tested the resource controller example, based on the client-server scheme, as given in section 4. We used one server and many number of client processes. We checked if the system violates the mutual exclusion property. SMC correctly indicated that mutual exclusion is not violated.

The following table shows the statistics for the above case. Each table entry contains two numbers. The first number corresponds to the case when only process symmetry is used, while the second number correspond to the case when both process symmetry and state symmetry are used; when a number is same as the preceding number we denote it by _. A * indicates that the program ran out of memory. The number of AQS states and the memory usage is same in both cases. The computation time is much smaller (by a factor of four) when state symmetry is also used. This indicates the power of state symmetry.

No. of client processes	100	150	200
AQS states	402/_	602/_	802/_
Comp. time (sec)	225/55	903/208	2502/549
Memory (kbyte)	662/_	1473/_	2604/_

Figure 6: Statistics for the Mutual exclusion property

For the above example, we have also checked if the following liveness property is satisfied: client 0 will eventually access the resource after sending a request for the resource. We checked it with strong fairness and, as expected, got the result that client 0 can be starved out. Here we used the second on-the-fly option where AQS nodes and edges are created on the fly during the product graph exploration and are stored after they are first constructed. Again we got the following statistics in this case. When state symmetry option is employed, the memory used (in addition to the computation time) is much smaller than when only process symmetry option is employed. When fairness is used the states and edges of the AQS are also stored. The edges also have associated permutations. Since AQS contains fewer edges when state symmetry is invoked (because only leader processes are executed), the memory usage is much smaller. It is also to be noted that the number of AQS nodes constructed is smaller than in the previous table; this is because the system constructed only a part of the AQS and found a fair computation that causes client 0 to starve. Note that when we had 200 clients the execution, using only process symmetry option, ran out of memory.

No. of client processes	100	150	200
AQS states	202/_	402/_	*/602
Comp. time (sec)	75/4	1640/42	*/160
Memory (kbyte)	1967/528	13475/3046	*/9058

Figure 7: Statistics for the Liveness property of the Resource Controller

8 Conclusions

In this paper we described our verification tool SMC. The main features of SMC are the following.

- It is a based on state space exploration with symmetry reduction. It utilizes both inter-state symmetry (called process symmetry) and intra-state symmetry (called state symmetry) to reduce the explored state space.
- It can be used to check for correctness under weak and strong fairness properties as specified by the user. It takes as input a concurrent program and a Buchi automaton specifying the incorrectness property and checks if there exists a fair computation of the program that is accepted by the automaton. Thus, SMC is capable of checking *safety* as well as all *liveness* properties expressible by finite state automata. It can also be used just for detecting deadlocks by not specifying the automaton component of the input.
- It is an on-the-fly model-checker which allows termination. It constructs the state space at the same time as it builds and explores the product graph.

The main motivation for designing the implemented system was to develop a model-checking system that checks for correctness under fairness and that uses symmetry to reduce the state space. We believe that SMC's input language is adequate for model-checking many concurrent systems. We concentrated on making SMC perform optimally on a special type of input architecture: on systems where the set of processes can be divided into modules, any permutation of the processes that preserves the module structure is an automorphism of the system. SMC was found highly effective on inputs of that special architecture and it performed fairly well on other types of inputs.

We used SMC on the link layer protocol of the IEEE Standard 1394. We found a deadlock in the protocol which was also reported earlier. We also analyzed certain liveness properties, more specifically the property that whenever a site a sends a data message it will eventually receive an acknowledgment for that message. We found that this property is not satisfied even under strong fairness and it requires stricter notion of fairness called transition fairness. The actual implementation of the physical layer should guarantee this fairness. To the best of our knowledge, ours is the first attempt at verifying liveness properties of the above protocol.

We plan to enhance the input language of SMC and also make SMC to be capable of handling various other types of symmetries,

Acknowledgments: We are thankful to the referees for their valuable comments and suggestions.

References

[1] Alur R., Henzinger T., Ho P. H., *Automatic Symbolic Verification of Embedded Systems*, Proceedings of the 14th Annual Real-Time Systems Symposium IEEE, pages 2-11, 1993.

[2] Aho, A. V., Hopcroft, J., Ullman, J. D.: *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] Aggarwal S., Kurshan R. P., Sabnani K. K.: *A Calculus for Protocol Specification and Validation.* in Protocol Specification, Testing and Verification III, H. Ruden, C. West (ed's), pp19–34, North-Holland 1983

[4] Bhat, G., Cleaveland, R., Grumberg, O.: *Efficient On-the-Fly Modelchecking for CTL.* International Conference on Logic in Computer Science, San Diego, California, 1995

[5] Clarke, E. M., Filkorn, T., Jha, S.: *Exploiting Symmetry in Temporal Logic Model Checking.* CAV93, LNCS **697** Springer-Verlag, 1993

[6] Cleaveland, R.: *Analyzing Concurrent Systems using the Concurrency Workbench, Functional Programming, Concurrency, Simulation, and Automated Reasoning.* pp129–144, LNCS **693** Springer-Verlag, 1993

[7] Cormen T. H., Leiserson C. E., Rivest R. L.: *Introduction to Algorithms.* MIT Press 1990

[8] Chandy K. M., Mishra J., *Prallel Program Design : A Foundation* Addison-Wesley 1989.

[9] Emerson E. A., Lei C. L., *Modalities for Model Checking: Branching Time Strikes Back*, Science of Computer Programming, Vol 8, 1997, pages 275-306.

[10] Emerson, E. A., Sistla, A. P.: *Symmetry and Model Checking.* CAV93, LNCS **697** Springer-Verlag, 1993

[11] Emerson, E. A., Sistla, A. P.: *Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach.* CAV95, LNCS **939** Springer-Verlag, 1995

[12] Francez,N., *Fairness.* Springer-Verlag, New-York, 1986

[13] Godefroid, P.: *Partial-Order Methods for the Verification of Concurrent Systems.* LNCS **1032** Springer-Verlag, 1996

[14] Gyuris, V., Sistla, A. P.: *On-the-Fly Model Checking under Fairness that Exploits Symmetry.* CAV97, LNCS **1254** Springer-Verlag, 1997; To appear in Formal Methods in System Design.

[15] *Handbook of Combinatorics. Vol. 1, 2.* editors: Graham, R. L., Grötschel, M., Lovász, L., Elsevier Science B.V., 1995

[16] Henzinger T. A., Nicollin X., Sifakis J., Yovine S., *Symbolic Model-Checking for Real-Time Systems*, Information and Computation 111, pages 193-244, 1994.

[17] Holzmann, G.J., Peled, D.: An Improvement in Formal Verification. Proc. FORTE 1994 Conference, Bern, Switzerland.

[18] Holzmann, G.J., Peled, D.: The State of SPIN. CAV96, LNCS **1102** Springer-Verlag, 1996

[19] Ip, C. N., Dill, D. L.: *Better Verification through Symmetry.* Formal Methods in System Design **9** 1/2, pp41–75, 1996

[20] *IEEE Standard for High Performance Serial Bus.* IEEE Standard 1394, Institution of Electrical and Electronic Engineers, 1995

[21] Jensen, K.: *Colored Petri Nets: Basic Concepts, Analysis Methods, and Practical Use, Vol2.* Analysis Methods, EATCS Monographs, Springer-Verlag, 1994

[22] Jha, S.: *Symmmetry and Induction in Model-Checking*, Ph. D. Thesis, Computer Science Department, Carnegie-Mellon University, 1996.

[23] *High-level Petri Nets: Theory and Application.* editors: Jensen, K., Rozenberg, G., Springer-Verlag, 1991

[24] Kuhne L., Hooman J., de Roever W., *Towards Mechanical Verification of Parts of the IEEE P1394 Serial Bus*, In Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia), June 1997.

[25] Kuang, X.: *Comparative Study of Deadlock-free Property on Software Architectures Specified Using OOADL and WRIGHT.* Master Thesis, University of Illinois at Chicago, 1997. Available at `http://www.eecs.uic.edu/~kxu/academic/academic.html`.

[26] Kurshan, R. P.: *Testing Containment of omega-regular Languages.* Bell Labs Tech. Report 1121-861010-33 (1986); conference version in *Reducibility in Analysis of Coordination*, pp19–39 LNCS **103** Springer-Verlag, 1987

[27] Kurshan, R. P.: *Computer Aided Verification of Coordinated Processes: The Automata Theoretic Approach.* Princeton Univerity Press, Princeton NJ, 1994.

[28] Luttik B., *Description and Formal Specification of the link layer of P1394*, In Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia), June 1997.

[29] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems—Specification*, Springer-Verlag 1992.

[30] McMillan, K.L.: *The SMV System, Draft.* Technical report, Carnegie Mellon University, 1992. Available at `http://www.cs.cmu.edu/afs/cs/project/modck/pub/www/smv.html`.

[31] Sighireanu, M., Mateescu, R.: *Validation of the link layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E–LOTOS.* INRIA Technical Report **3172**, 1997

[32] Tarjan, R. E., *Depth first search and Linear Graph Algorithms*, SIAM J. Computing 1:2, 1972, pp 146-170.

[33] Valmari, A.: *A stubborn attack on state explosion*, The Second International Conference on Computer Aided Verification (CAV90), New Brunswick, NJ, 1990

9 Appendix

9.1 Syntax for the Program

Below we give the formal syntax of the input language of SMC. For the sake of brevity, we did not give the production rules corresponding to the non-terminals `<name>, <value>, <variable>` and `<condition>`. The production rules corresponding to all other non-terminals are presented. The non-terminal `<name>` is an identifier; `<value>` is a sequence of digits; `<variable>` defines a program variable, it is an identifier followed by a list of integers, separated by comma, and enclosed with in `[]`. The `<condition>` non-terminal defines an expression involving arithmetic operations + and *, and boolean operations. Such an expression can be considered as an integer expression or as a boolean expression (as in the C language). The condition part can involve quantifiers over index variables.

9.2 Syntax for the automaton

Below we give the BNF for the automata specification.

9.3 Checking Equivalence of States

The following algorithm is the pseudo-code for phase 1 of the equivalence checking algorithm. For each state $u = s, t$, for each process k, it maintains the following data structures. $New\_Sig(k, u)$, $Old\_Sig(k, u)$, respectively, denote the newly computed signature and the old signature of process k in state u. Similarly $New\_Set(u)$, $Old\_Set(u)$ respectively denote the new and old sets of signatures in state u. For each state u, $New\_Processes(u)$ is an array indexed by the signatures so that $New\_Processes(u)(S)$ denotes the set of processes whose new signature in state u equal S. Similarly $Old\_Processes(u)(S)$ denotes the set of processes whose old signature in state u equal S.

The algorithm works as follows. In statements 1 and 2, it computes the initial signatures and processes having the signatures as described in sub-section 5.3. In statement 5, the new signatures are calculated (i.e. signatures in the $(n+1)st$ iteration are calculated from the signatures in the n^{th} iteration by using the formulas given in subsection 5.3). Here the set of new signatures given by $New\_Set(s)$ and the processes having these signatures, given by $New\_Processes(s)$, are also calculated. In statement 11, it is checked if the signature sets in s and t are equal and the number of processes having a given signature in s is same as that in t. If the check fails then the states are not equivalent and a no answer is returned. In statement 12, the termination condition is checked (i.e. if there is no refinement). Here it is checked if the new partition on the set of processes is same as the old one; this indicates a fix-point with respect to state s(it is not difficult to see that this fix-point check can be done with respect to s or t as both conditions will be satisfied in the same iteration). If the fix-point is reached then an YES answer together with the set of signatures and the processes having these signatures in s and t are returned.

Function Phase1(s, t)

```
1 For each process k, and each u ∈ {s,t},
      Compute Old_Sig(k,u)
2 Compute Old_Set(u) and Old_Processes(u)
      for each u ∈ {s,t}

3 While True
4     { For each u ∈ {s,t}, New_Set(u) := ∅

5     For each process k and each u ∈ {s,t}
6         { Compute New_Sig(k,u)
7         If New_Sig(k,u) ∉ New_Set(u)
8                 add New_Sig(k,u) to New_Set(u)
9                 New_Processes(u)(New_sig(k,u)) := {k}
          Else
10                insert k in to the set New_Processes(u)(New_Sig(k,u))
          }
11    If New_Set(s) ≠ New_Set(t) or there is some
          S ∈ New_Set(s) such that
                |New_Processes(s)(S)| ≠ |New_Processes(t)(S)|
      then Return(NO)

12    If (for each S ∈ New_Set(s) there exists an
          S' ∈ Old_Set(s) such that
                  New_Processes(s)(S) = Old_Processes(s)(S'))
      then Return (YES,New_Set(s), New_Processes(s),
                  New_Processes(t))

13    For each u = s,t, for each k,
          Old_Set(u) := New_Set(u)
          Old_Processes(u) := New_Processes(u)
      }
```

```
<program>  : Program <commands>

<commands> : <commands> <command>
         | <command>

<command>  : Module  <name> = <value> ;
         | <name> [ <some_module_names> ] = <value> ;
         | <ind_var> : <module>
         | <ind_var> ;

<some_module_names> :
         | <module_names>

<module_names> : <name>
         | <name>,  <module_names>

<ind_var>  : <name>
         | <name> 'of' <name>

<module>   : { <module_commands> }
         | <module_command>

<module_commands> : <module_commands> <module_command>
         |

<module_command> : <condition> -> <actions> ;

<actions>  : <action>
         | <action> , <actions>

<action>   : <variable>  = <condition>
         | ALL ( <ind_var> : <actions> )
```

Figure 8: The Input Language

```
<automata>        : InitialState <name> ; <a_commands>

<a_commands>      :
                | <a_command> ; <a_commands>

<a_command>      : FinalState
                | FinalState <a_fin_states>
                | Tracking
                | Tracking <a_trackings>
                | <a_transition>

<a_fin_states>    : <name>
                  | <name> , <a_fin_states>

<a_trackings>     : <a_tracking>
                  | <a_tracking> , <a_trackings>

<a_tracking>      : <number> 'of' <name>

<a_transition>    : <condition> -> <name> : <name>
```

Figure 9: The Automata Declaration Language