Under consideration for publication in Formal Aspects of Computing

Verifying the IEEE 1394 FireWire Tree Identify Protocol with SMV

Viktor Schuppan and Armin Biere

Computer Systems Institute
Department of Computer Science
Swiss Federal Institute of Technology Zurich, Switzerland

Abstract. This case study contains a formal verification of the IEEE 1394 FireWire Tree Identify Protocol. Crucial properties of finite models of the protocol have been validated with state-of-the-art symbolic model checkers. Various optimization techniques were applied to verify concrete and generic configurations.

Keywords: IEEE 1394 FireWire, Formal Methods, Protocol Verification, Model Checking.

1. Motivation

Traditional ways of validating and verifying the functional correctness of software include testing/simulation and theorem proving [KMM00, Sch01]. Testing is based on a collection of test cases, correctness is assured for each test case. The number of test cases may grow exponentially with the number of input variables. Therefore, it is usually impossible to cover all potential behavior in a test suite. No knowledge of a specialized formalism is required. On the other hand, with theorem proving, correctness of software can be verified formally. Every potential behavior is covered. However, in-depth knowledge and a lot of of experience in the use of the methodology is necessary.

Model checking combines some of the advantages of both testing and theorem proving. Depending on the number of parameters left unspecified in the model a configuration corresponds to either a single or a large number of test cases which are verified in a single run of the model checker. By leaving all parameters unspecified all possible behaviors can be covered. However, with more free parameters search space grows and thus the run time increases. The use of a model checker requires only moderate knowledge of the underlying theory. In the past, model checking has successfully been applied to several case studies in industry, see, e.g. [CGH+93], [Ben01], and [TWC01].

BDD-based symbolic model checking [BCM92] can handle systems with more than 10^{20} states. Still, for verification problems typically found in industry BDDs may grow too large for the hardware available today. Symmetry reduction [CFJ93], [ID96], [ES93] is one approach to reduce the state space to be explored by exploiting symmetries inherent in the problem. Bounded model checking [BCCZ99] provides another potential solution to the above mentioned problem but can only find counterexample traces up to a user-specified length.

Correspondence and offprint requests to: Viktor Schuppan, Computer Systems Institute, ETH Zurich, 8092 Zurich, Switzerland. E-Mail: Viktor.Schuppan@inf.ethz.ch

SMV[1] [McM93] is probably one of the most widely used symbolic model checkers. Several variants of the original version exist that still understand models in the original language but provide improved engines [YSMV] and advanced features and algorithms [CSMV], [CCG+02].

We have modeled and verified the Tree Identify Protocol (TIP) of the IEEE 1394 (FireWire) standard[2] [1394-1995], [1394-2000], proposed as a case study for the application of formal methods [MRS02], with *SMV*. Starting with specific network configurations we moved to more flexible models that include all legal topologies with a given number of nodes and ports. We applied symmetry reduction and bounded model checking to evaluate what gains in performance and capacity can be achieved.

In the next section, techniques and tools for symbolic model checking are introduced. Symmetry reduction and bounded model checking are explained. In Sect. 3 we explain the models and the specifications used for our verification of the protocol. Section 4 gives the results. These are discussed in Sect. 5 and in Sect. 6 we draw a brief conclusion.

2. Model Checking

In this section we give a brief overview of model checking. Symmetry reduction and bounded model checking are discussed. A simplified model of a node in a FireWire system is used to illustrate the techniques and to introduce the language of the model checkers used. For an in-depth coverage consult recent textbooks [CGP99], [BBF+01].

2.1. Symbolic Model Checking with SMV

Model checking was introduced independently by Clarke and Emerson [CE81] and by Quielle and Sifakis [QS82] in 1981 as a technique for verification of finite state concurrent systems. Systematic state space exploration is used to check a system against its specification. The specification is given as a number of formulae in either branching or linear time temporal logic (see, e.g., [Eme90] for an introduction). The system itself is represented as a labeled state-transition graph or *Kripke structure*. A Kripke structure is a tuple $M = (S, I, T, l)$ consisting of a finite set of states S, a set $I \subseteq S$ of initial states, a transition relation $T \subseteq S \times S$, and a labeling function $l : S \rightarrow 2^A$ where A is a set of atomic propositions. A *path* $\pi = (s_i)$ in a Kripke structure is a finite or infinite sequence of states with $s_i \in S$ and $T(s_i, s_{i+1})$ for $0 \leq i < |\pi|$. When a model checker detects that a system does not satisfy its specification it can produce a counterexample to help the user track down the error.

The first implementations of the model checking algorithm represented the state-transition graph explicitly. Sets of states and transition relations can often be represented more compactly by stating their characteristic functions instead of enumerating their elements explicitly. Verification is then performed by appropriate calculations on the corresponding functions. For example, given a predicate $f_I(s)$ evaluating to true iff s is an initial state and another predicate $f_T(s, s')$ that is true iff s' is a successor state of s, the set of states reachable in one step from the set of initial states is given by $f_1(s) \equiv \exists i \in S. f_I(i) \wedge f_T(i, s)$. Similarly, reachability of an error state can be checked by iteratively calculating $f_1(s),\ f_2(s) \equiv \exists t \in S. f_1(t) \wedge f_T(t, s),\ \ldots$ and intersecting intermediate results with the set of error states until reachability is proved or a fixed point is reached.

This idea was proposed first by Burch, Clarke, and McMillan [BCM92] using Bryant's *reduced ordered binary decision diagrams* [Br86] as a canonical representation for boolean functions. With the new representation, McMillan's *SMV* [McM93] could verify systems with more than 10^{20} states, reaching more than 10^{100} states today. However, for the verification of larger systems BDD-based symbolic model checking requires careful ordering of the input variables to avoid space blow-up of the BDDs. In some cases no space-efficient ordering can be found [Br91]. *SMV* has been used in a number of case studies. See, for example, [CGH+93], [CCW+95]. Its source code is available at [SMV].

Modeling State Machines in SMV

In *SMV*, a system is modeled as a state machine. The input language is similar to common hardware description languages. The properties to be verified are given as temporal formulae in computation tree logic (CTL) [Eme90], [CE81]. The state of the system is determined by the values of a collection of state variables. Available types include booleans, integer ranges, sets of user-defined symbolic values as well as arrays. Compound types can be represented by

[1] In the sequel, *SMV* refers to the original version of *SMV* distributed by CMU.
[2] The term "standard" without further explanation is used as synonym for the IEEE 1394 (FireWire) standard [1394-1995, 1394-2000].

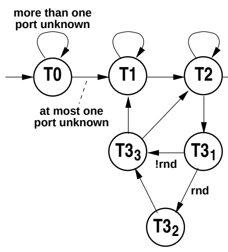

Fig. 1. Simplified state transition diagram of our model of the FireWire Tree Identify Protocol (TIP).

```
MODULE node
VAR
  s: {T0, T1, T2,
      T3_1, T3_2, T3_3, S0};
  p: array 0..2 of port;
  rnd: boolean;

ASSIGN
  init(s) := T0;
  next(s) := case
    s = T0 &
      ((!(p[0].role = unknown) &
        !(p[1].role = unknown)) |
       (!(p[0].role = unknown) &
        !(p[2].role = unknown)) |
       (!(p[1].role = unknown) &
        !(p[2].role = unknown))): T1;
    s = T1: {T1, T2};
    s = T2: {T2, S0, T3_1};
    s = T3_1 & rnd: T3_2;
    s = T3_1 & !rnd: T3_3;
    s = T3_2: T3_3;
    s = T3_3: {T1, T2};
    1: s;
  esac;
```

Fig. 2. Model of a node in *SMV*.

```
scalarset PORT 0..2;
module node() {
  s: {T0, T1, T2, T3_1, T3_2, T3_3, S0};
  p: array PORT of port;
  rnd: boolean;
  u: array PORT of array PORT of boolean;
  v: array PORT of boolean;

  init(s) := T0;
  default {next(s) := s;}
  in switch(s) {
    T0: {
      forall(i in PORT) {
        forall(j in PORT) {
          u[i][j] := (i ~= j) ->
            ~((p[i].role = unknown) &
              (p[j].role = unknown));
        }
        v[i] := &u[i];
      }
      if (&v) next(s) := T1;
    }
    T1: next(s) := {T1, T2};
    T2: next(s) := {T2, S0, T3_1};
    T3_1: if (rnd)
            next(s) := T3_2;
          else if (~rnd)
            next(s) := T3_3;
    T3_2: next(s) := T3_3;
    T3_3: next(s) := {T1, T2};
  }
}
```

Fig. 3. Model of a node in *Cadence SMV* using scalarsets.

hierarchical modules. The set of initial states and the transition relation between the states are defined by assignments to the initial and next state values of the state variables. Invariants, constraints on the transition relation, and fairness conditions may be added as boolean formulae.

A system is structured into a number of modules. Each module can be instantiated multiple times and may contain instances of other modules. *SMV* allows for both synchronous and asynchronous execution of the model, but has no provisions for expressing continuous time constraints. Communication takes place via shared variables.

As an example Fig. 2 shows a simplified part of our model of a node of a FireWire system. It implements the state machine of the tree identify protocol. The corresponding state transition diagram is given in Fig 1. While the set of states includes all states of our model, most conditions on the state transitions have been abstracted to a non-deterministic choice to simplify the presentation. Details on the meaning of states and transitions can be found in Sect. 3 and [MRS02], [1394-1995], [1394-2000].

The state of each instance of `node` is defined by 3 variables. The variable `s` stores the node's current state in the TIP state machine. Each node has 3 ports declared in a separate user-defined module `port`. Boolean `rnd` is used as random bit in the protocol.

Execution of the protocol starts in the initial state $T0$. A node can only move from $T0$ to $T1$ if the role of at most one port is *unknown*. Boolean *and*, *or*, and *not* are expressed with &, `|`, and `!`, respectively. As all combinations have to be listed explicitly the length of the expression for the transition from $T0$ to $T1$ is quadratic in the number of ports. In state $T3_1$ the value of `rnd` decides whether a node moves to $T3_2$ or $T3_3$. `rnd` is never assigned a value in the model – this corresponds to a non-deterministic choice. The branches of the case-statement are checked in their order of appearance, the first that evaluates to true is taken. The last branch ensures that a node remains in its current state if no other condition is true.

Variants

Based on the original version of *SMV* several variants have been developed. *NuSMV* [CCG+02] is a reimplementation of the original version developed by CMU and IRST. It provides additional interfaces and allows specifications in both CTL and linear temporal logic (LTL, [Eme90], [CGP99]). Further improvements concerned modularity of the design and robustness of the implementation. The latest version includes a bounded model checking engine and past time operators for LTL. Yang [YSMV] independently rewrote large parts of *SMV*. While preserving the features of the original version, in particular the input language, Yang's implementation gives significant speed-up in many cases.

McMillan continues to improve *SMV* at Cadence Berkeley Labs. A research version [CSMV] is freely available for non-commercial purposes. In our work, we used the following additional features of *Cadence SMV*:

- *Enhanced input language*. Modules can be separated into different files and included via C-like include directives. Most other C-preprocessor directives are also available. For-loops allow simple iterations over array elements. Indices of arrays can be array elements as well. Specifications are usually expressed in LTL if the enhanced input language is used.
- *Assume guarantee reasoning*. Assertions can be used in the proof of other assertions. Assertions marked as assumptions do not need to be proved.
- *Symmetry reduction*. Scalar types can be defined as scalarset enabling symmetry reduction on these types.
- *Bounded model checking*. By default an interface to *zChaff* [ZMM+01] is provided. Other SAT-solvers can also be integrated.

Further improvements not used in this case study include refinement verification, data type reduction, and induction.

2.2. Symmetry Reduction

Symmetry is found in many systems considered for the application of model checking. Examples are several instances of a worker thread in a multi-threaded application or nodes of the same type in a network. In both cases, permuting objects (i. e., threads or nodes) and references to those objects in a consistent way results in states with equivalent behavior. Symmetry reduction [CFJ93], [ID96], [ES93] exploits this fact to reduce the state space to be searched in model checking by selecting only a subset of representatives for such sets of states. In some cases this reduction may have an exponential benefit.

More formally [CFJ93], consider a Kripke structure $M = (S, I, T, l)$. A group of permutations P on S is called a *symmetry group* of M if each permutation $\sigma \in P$ preserves T: $\forall s_1 \in S, \forall s_2 \in S : (s_1, s_2) \in T \Leftrightarrow (\sigma(s_1), \sigma(s_2)) \in T$. The *orbit* of a state $s \in S$ is the set of states $\theta(s) = \{\hat{s} \in S \mid \exists \sigma \in P\colon \sigma(\hat{s}) = s\}$. An atomic proposition a is called *invariant* under P, if the set of states labeled with a is closed under the application of permutations in P. That is $\forall s \in S, \forall \sigma \in P : a \in l(s) \Leftrightarrow a \in l(\sigma(s))$. Clarke et. al. show [CFJ93] that, if all propositions of a formula h are invariant under P, it suffices to check $M_P \models h$ instead of $M \models h$, where M_P is the *quotient model* of M: $M_P = \{S_P, I_P, T_P, l_P\}$ with $S_P = \{\theta(s) \mid s \in S\}$, $I_P = \{\theta(s) \mid s \in I\}$, $T_P = \{(\theta(s_1), \theta(s_2)) \mid \exists s_3 \in \theta(s_1), \exists s_4 \in \theta(s_2) : (s_3, s_4) \in T\}$, and l_P : $l_P(\theta(s)) = l(any(\theta(s)))$

Trying to find symmetries automatically without user guidance has not found widespread application, probably because the problem seems to be almost as expensive as the calculation of reachable states. *Scalarsets* were introduced by Ip and Dill [ID96] in *Murphi* [DDH+92] and implemented in *Cadence SMV*. As a fully symmetric data type they enable the user to specify where symmetry reduction can be applied. Scalarsets are finite subranges with test for equality as the only legal operation between two elements of a scalar set. More operations are possible when applied to all elements of a scalarset. The elements of a scalarset can be arbitrarily permuted. Elements of arrays indexed with a scalarset type may also be permuted as long as all references are updated accordingly. In this case, only one particular ordering of a given set of array elements needs to be verified.

Model of a Node with Symmetry Reduction

Figure 3 shows our model of the TIP state machine in the enhanced input language of *Cadence SMV*. To enable symmetry reduction on the ports of a node, a scalarset type is defined for port indices. Access to ports `p` must now ensure that symmetry is not broken. Therefore, two nested `forall`-loops are used to determine whether there exists a pair of different ports whose role is unknown. The result is collected using two- and one-dimensional arrays `u` and `v` with the &-operator being overloaded to calculate the boolean *and* of all elements of a vector. `->` and `~` denote boolean implication and negation. Note also, that the expression for the transition from $T0$ to $T1$ can now be formulated independent of the specific number of nodes. The `case`-statement with arbitrary boolean expressions at the branches in Fig. 2 is replaced by a `switch`-statement ranging over all states and an `if`-statement within the branches where necessary. The `default`-statement is executed if no assignment takes place within the `switch` part.

2.3. Bounded Model Checking

In contrast to BDD-based algorithms, SAT-based symbolic model checking or *bounded model checking* as proposed by Biere et. al. [BCCZ99] is typically independent of a good variable order. In addition, counterexamples are generated faster and shortest counterexamples can be found.

A bounded model checker translates a Kripke structure and a specification into a boolean formula which is satisfiable iff a counterexample of a user-specified length k exists. This formula is then passed to an off-the-shelf SAT solver, for example *SATO* [Zha97], *Prover* [Bor97] or *zChaff* [ZMM+01]. If the formula is satisfiable a counterexample can be generated from the satisfying assignment.

Shtrichman [Sht00] proposed several strategies to optimize SAT solvers for bounded model checking. In several case studies his strategies outperformed traditional BDD-based symbolic model checking. Bounded model checking has also successfully been applied in industrial verification projects [CFF+01], [BLM01].

Bounded Model Checking of the Node Model

Bounded model checking can best be explained with an example. Consider the model of a FireWire node in Fig. 2 again. The system's current state is given by variables s, p, and rnd. We assume that the predicates representing the initial state and next state relations of p are defined by I_p and T_p. In addition, we do not refer to individual bits of a state variable but use compound expressions as in $s = T0$ as propositional formulae. Primed variables refer to the value of a variable in the next state.

Initially, a node is in state $T0$, rnd is unconstrained, and p is in a valid initial state: $I(s,rnd,p) \;\equiv\; s = T0 \wedge I_p(p)$. Similarly, the transition relation T can be expressed as the conjunction of the next state relations $step_s$, $step_{rnd}$, and T_p:

$$\begin{array}{lcll}
u(p) & \equiv & p[0].role \neq unknown \wedge p[1].role \neq unknown & \vee \\
 & & p[0].role \neq unknown \wedge p[2].role \neq unknown & \vee \\
 & & p[1].role \neq unknown \wedge p[2].role \neq unknown & \\
step_s(s,p,rnd,s') & \equiv & s = T0 \Rightarrow (\neg u(p) \wedge s' = T0 \vee u(p) \wedge s' = T1) & \wedge \\
 & & s = T1 \Rightarrow (s' = T1 \vee s' = T2) & \wedge \\
 & & s = T2 \Rightarrow (s' = T2 \vee s' = S0 \vee s' = T3_1) & \wedge \\
 & & s = T3_1 \Rightarrow (rnd \wedge s' = T3_2 \vee \neg rnd \wedge s' = T3_3) & \wedge \\
 & & s = T3_2 \Rightarrow s' = T3_3 & \wedge \\
 & & s = T3_3 \Rightarrow (s' = T1 \vee s' = T2) & \\
step_{rnd}(rnd,rnd') & \equiv & true & \\
T(s,s',rnd,rnd',p,p') & \equiv & step_s(s,s') \wedge step_{rnd}(rnd,rnd') \wedge T_p(p,p') &
\end{array}$$

To verify the liveness property $AF\ (s = S0)$ a formula is constructed which is satisfiable iff a counterexample exists: $EG\ (s \neq S0)$. Then the set of all valid paths of length k (i. e., $k+1$ states) is given by $M_k \equiv I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$. Every witness for $EG\ (s \neq S0)$ has infinite length. Therefore, a witness with $k+1$ states must contain a loop: $L_k \equiv \bigvee_{i=0}^{k} T(s_k, s_i)$. Finally, $s \neq S0$ must hold for every state: $P_k \equiv \bigwedge_{i=0}^{k} s_i \neq S0$. Now, $W_k \equiv M_k \wedge L_k \wedge P_k$ can be passed to a SAT solver. In the example, a counterexample will be produced for any $k \geq 0$.

3. Model

We have built a number of models to evaluate performance and capacity of some variants of *SMV*. We started with an implementation for the version of *SMV* from CMU. A reimplementation was done to test symmetry reduction with *Cadence SMV*. Finally, minor modifications enabled us to perform bounded model checking. After giving some terminology we describe the implementation and specification for *SMV*. Then, we state the differences to our original version made for model checking and bounded model checking with *Cadence SMV*.

3.1. Terminology

Our model [URL] of a FireWire system is based on the structure as shown in sections 3.7.3.1.2 and E.3.2 of the FireWire standard [1394-1995]. A specific *configuration* of a system is defined by

- the number of *nodes*,
- the number of *ports* available at each node,
- the *topology*, given by the interconnection of the nodes in the network, and
- the value of the *force-root*-flag of each node.

Each port is either connected to other ports or inactive. A more detailed explanation of the terms used can be found in the introductory paper [MRS02] of this issue.

3.2. Implementation

The implementation of a node is similar to the Tree ID state machine in [1394-2000]. A state variable holds the state of each node in the state machine. The set of states includes states $T0$, $T1$, and $T2$ of the Tree Identify Protocol as described in the standard. State $T3$ is refined (see below) and the first state of the self-identify phase, $S0$, is added as an end state. See Fig. 1 for a state-transition diagram. Note, that most conditions on the transition relation have been abstracted to a non-deterministic choice.

In our experiments we only use synchronous execution of the model. Because of technical difficulties we were not able to include some features of the FireWire TIP in an asynchronous model that can be processed by the original or Yang's version of *SMV*. To allow for a fair comparison with *Cadence SMV* we also used a synchronous version there. From that, an asynchronous model can be constructed with some additional effort.

For low-level communication, IEEE 1394 uses two signals with three different states. We abstract from this low-level line-based encoding scheme and use the line states given in Tables 4-27 and 4-28 in [1394-1995]. Not all states are relevant for the tree identify protocol [1394-1995]. In our implementation each port has two state variables which take the line state to be transmitted and received as their values. Invariants ensure that input and output ports match.

For the random choice of bits during resolution of root contention [1394-2000], non-determinism is used in the transition relation. A fairness condition ensures that the choices will differ after an arbitrary but finite number of steps.

3.3. Timing

Timing is an important issue at several points in the Tree Identify Protocol:

- timers are used to detect *time-outs* in the protocol which indicate a cycle in the configuration,
- a node may intentionally be delayed in its first state to increase its chance of becoming root (called *force-root*),
- *root contention* is resolved by time delays of different length.

SMV does not offer constructs to express continuous time constraints. Therefore, we use one counter per node to model time-out and force-root conditions. The counter is incremented each time-step as long as the node is in state $T0$. If force-root is set the node cannot execute the transition from $T0$ to $T1$ until its counter has reached a certain limit l_1. If the counter exceeds another limit $l_2 > l_1$, a flag indicating time-out is set. In contrast to the standard, both limits depend on the number of nodes in a configuration. This is to minimize the state space while preserving the desired effect in the protocol. For the resolution of root contention in the standard, nodes have to wait different amounts of time. We implement this by making each of the two nodes non-deterministically choose one of two paths of different length in the state machine. For this purpose, state $T3$ is split into 3 sub-states. $T3_1$ and $T3_3$ serve as entry and exit states. $T3_3$ is reachable from $T3_1$ either directly on or on a path of length two via $T3_2$.

3.4. Parameters

The size of the state space to be searched and, thus, the amount of time and memory needed for the verification depends on the number of free variables in the model. For each configuration there are two groups of variables that can either be specified by the user or left undetermined: the network topology and the force-root flag. This gives 4 combinations of (un-)determination[3] in our model:

- specified topology, force-root set to 0, or fully determined
- specified topology, force-root unspecified, or force-root undetermined
- unspecified topology, force-root set to 0, or topology undetermined
- unspecified topology, force-root unspecified, or fully undetermined

Topology

The topology of a FireWire system is given by the interconnection of nodes in the network. Information about the topology is kept in a global table. If only a certain topology is to be checked the user can initialize the table with that topology. Otherwise, all configurations with legal topologies that can be formed by the given number of nodes using up to the specified maximum number of ports are verified. In both cases, the topology is fixed during the run of the protocol. A topology is called *legal* iff

1. it is *sound*, i. e. for each pair of input and output lines destination and source ports match,
2. it is *connected*, and
3. it contains *no cycles*.

These conditions are ensured by invariants. Condition 1 is straight-forward. Reachability in conditions 2 and 3 is checked by explicitly enumerating all possible paths between any two nodes and checking the validity of each path.

Force-Root Flag

For each configuration another parameter remains to be chosen: the force-root flag of each node. The information is kept in a local variable of each node. It also remains fixed throughout the protocol. If more than one node has its force-root flag set, none of them might become root.

3.5. Specification

The two most important properties for the protocol are stated in the problem description [MRS02]:

1. Eventually, at least one node becomes root.
2. At any time, at most one node is root.

We have specified three additional properties to ensure that each node reaches a well defined, safe state at the end of the protocol. It is required that

3. Eventually, every node is and remains in state $S0$.
4. Eventually, the role of the peer node at each connected port is and remains determined.
5. Eventually, each link is and remains in state idle.

Depending on the particular configuration, further requirements are added to specify the desired outcome of the election protocol.

There are certain exceptions to requirements 1 – 5. For example, if the user configures nodes into a cycle a time-out should occur. In this case some requirements stated above will not be met although a legal end-state in this particular configuration is always reached.

The specification can be extended to include exceptions and potential problems into the given requirements. Now the model checker is ready to report whether the (extended) requirements are met or further problems are present. The presence of known exceptions or problems in a given configuration can still be checked by separate specifications. For instance, we include a time-out specification in our implementation. A typical requirement looks as follows:

```
AF ((AG in_state_S0) | timeout)
```

[3] *Undetermination* in our terminology corresponds to a non-deterministic choice at the start of the execution of the model.

This states that all nodes finally arrive and remain in state *S*0 or a time-out occurs. If a timeout actually has occurred (depending on the current configuration) can be seen by verifying one more clause: `AG !timeout`. It is also possible to add a requirement stating that if a configuration is cyclic then a timeout will be produced.

3.6. Generation of the Model

Neither preprocessor directives nor for-loops are available in the input language of the original *SMV*. To avoid the first problem, we used the C-preprocessor *cpp*. Lack of for loops however makes it necessary to generate separate input files for each configuration. In addition, the exponential growth in the number of possible paths in the network leads to a blow-up in the file size for more than a handful of nodes.

3.7. Advanced Features of Cadence SMV

To use advanced features of *Cadence SMV* large parts of the model had to be rewritten. Node and port indices are defined as scalarset types to enable symmetry reduction. Preprocessor directives and nested for loops allow us to parameterize the model and the specification in the numbers of nodes and ports. Now it suffices to have a single generic input file with only few parameters to be filled in for a particular configuration.

Many parts of the model can be formulated more elegantly. The mapping from output to input lines can be stated in *Cadence SMV* using for loops and nesting of array indices as essential features within 5 lines of code. The corresponding statement in *SMV* takes more than a page for a configuration with 3 nodes and 3 ports. To check whether a topology is connected and contains no cycle the transitive closure of the transition relation is computed instead of listing all paths explicitly. The blow-up in the file size is avoided.

In *Cadence SMV*, LTL is used instead of CTL. The example from the previous section now reads (remember that LTL formulae are implicitly quantified over all paths): `F((G in_state_S0)|timeout)`. Note that the meaning of both formulae is different. The semantics of the LTL version matches the standard as we understand it. However, as it can be shown that the CTL formula actually represents a stronger statement, which suffices for our purposes.

Assume-guarantee reasoning is used instead of invariants to ensure that the desired properties only need to be proven for fair executions of legal configurations. If a certain user-specified topology is to be verified, the assumptions for legal configurations are dropped, since they can actually be verified.

3.8. Bounded Model Checking with Cadence SMV

In bounded model checking only paths up to a user-specified length *k* are explored. This requires a change in the mechanism for random bit selection during the resolution of root contention. The non-deterministic choices of the two contending nodes are now forced to differ after a fixed number *l* of iterations through the sub-protocol. To derive the path length *k* used for bounded model checking we manually calculated the maximum number of time steps it takes for a legal configuration to complete a run of the protocol or for a timeout to occur in a cyclic configuration as follows:

	# time steps until force-root has passed
$+$	# time steps until at most two root candidates remain or a cycle has been detected
$+$	# time steps for *l* non-successful runs through root contention resolution
$+$	# time steps for one successful run through root contention resolution
$+$	# time steps for root to complete protocol
$+$	safety margin of 2.

Bounded model checking can not fully verify the correctness of this protocol because execution sequences of arbitrary length exist. However, our goal was only to show how expensive bounded model checking for certain examples may be, even if completeness is not that important.

4. Results

The specification has been verified for a number of topologies with synchronous execution. For all legal topologies we could prove the specifications to be valid. In topologies containing a cycle the cycle was detected. We used configurations with 2 to 6 nodes and 2 to 4 ports without specifying a topology. In addition, user-specified topologies with 3

Table 1. Results for Yang's version of the original SMV with fixed topology

Configuration		*det.*			*frn.*		
nodes	**ports**	# states	time [s]	mem [Mb]	# states	time [s]	mem [Mb]
2	3	2ˆ15.5	0.2	186.3	2ˆ18.1	0.3	186.6
3	3	2ˆ26.7	0.3	186.9	2ˆ29.7	0.5	188.1
5	3	2ˆ41.0	0.8	192.0	2ˆ46.2	1.8	197.7
6	4	2ˆ54.2	1.5	197.6	2ˆ60.7	3.8	209.2
6c	4	2ˆ51.2	2.1	201.5	2ˆ57.9	2.2	201.2
10	3	2ˆ66.9	2.6	210.7	2ˆ77.2	64.4	442.8

Table 2. Results for Yang's version of the original SMV with unspecified topology

Configuration		*topn.*			*n.*		
nodes	**ports**	# states	time [s]	mem [Mb]	# states	time [s]	mem [Mb]
	2	2ˆ15.6	0.3	186.2	2ˆ15.7	0.4	186.8
2	3	2ˆ25.3	1.2	191.2	2ˆ27.6	1.3	192.3
	4	2ˆ35.6	4.7	215.1	2ˆ38.1	5.6	220.9
	2	2ˆ26.8	1.3	193.0	2ˆ29.8	3.5	204.3
3	3	2ˆ42.0	8.1	241.5	2ˆ45.0	22.5	303.5
	4	2ˆ57.9	79.9	442.9	2ˆ60.9	544.4	819.4
	2	2ˆ30.8	12.7	257.4	2ˆ34.9	90.1	442.3
4	3	m.o.	811.2	949.8	m.o.	8368.6	965.3
	4	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.
	2	2ˆ46.8	174.5	809.9	2ˆ52.0	7585.2	959.3
5	3/4	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.

ports (apart from **(iv)** with a cycle – 4 ports are needed here) were used. Topologies **(i)** – **(iii)** and **(v)** are taken from the workshop version [SB01] of this paper, **(iv)** is the topology Calder and Miller have used in their case study [CM01] with *SPIN* [Hol91]:

(i) two nodes
(ii) three nodes forming a chain
(iii) tree with five nodes configured as in sections 3.7.3.1.2 and E.3.2 of [1394-1995] (with 3 ports available per node)
(iv) tree with six nodes as used by Calder and Miller in [CM01] (with cycle, marked "c", and without a cycle)
(v) tree with 10 nodes

Where possible we verified all combinations of (un-)determination in our model. In Tables 1 - 4 fully determined configurations are marked *det.*, force-root undetermined configurations are marked *frn.*, topology-undetermined configurations have *topn*, and fully undetermined configurations have *n.* respectively. Combinations not used are marked "-". For each configuration we list the user time in seconds as reported by the model checkers and the memory allocated in megabytes as reported by the tool *memusage*. For all experiments we used a wall clock limit of 6 h. Memory out and time out are indicated by "m.o." and "t.o.".

We spent several days to construct a good variable order. Reversing the default variable ordering resulted in considerable runtime improvements in *Cadence SMV*. Further optimizations based on dynamic and additional manual variable reordering did not enable us to verify larger configurations.

4.1. SMV

Because of its superior performance, we used Yang's implementation [YSMV] for the experiments in this section. We enabled forward model checking without changing the default variable order. The experiments were conducted on a Pentium III at 800 MHz with 1.5 Gbytes main memory running Linux 2.2.19. In addition to user time and memory, tables 1, 2 also list the number of states reachable for each configuration. We could not handle configurations with 6 nodes and unspecified topology within these time and memory bounds.

Table 3. Results for Cadence SMV with symmetry reduction

Configuration		*topn.*		*topn. symm.*		*n.*		*n. symm.*	
nodes	**ports**	time [s]	mem [Mb]	time [s]	mem [Mb]	time [s]	mem [Mb]	time [s]	mem [Mb]
2	2	2.1	3.0	2.1	3.0	2.3	3.7	2.3	3.7
	3	5.6	4.7	3.6	4.5	5.8	4.6	4.1	4.8
	4	10.9	8.2	6.0	13.0	12.2	8.6	6.5	5.5
3	2	5.6	6.6	4.6	5.3	48.9	17.9	33.8	15.2
	3	83.9	56.4	45.1	59.5	933.6	160.9	329.1	112.9
	4	1993.8	1408.1	1521.6	418.7	m.o.	m.o.	3524.4	1149.0
4	2	254.1	119.6	139.5	70.7	2154.9	357.9	2191.8	242.0
5	2	8070.5	1269.7	1272.3	598.6	m.o.	m.o.	m.o.	m.o.

Table 4. Results for bounded model checking with Cadence SMV

Configuration		*det.*						*n.*					
		$l = 0$			$l = 2$			$l = 0$			$l = 2$		
nodes	**ports**	k	time [s]	mem [Mb]	k	time [s]	mem [Mb]	k	time [s]	mem [Mb]	k	time [s]	mem [Mb]
2	3	20	91.4	14.0	28	216.1	20.8	20	615.4	18.9	28	2808.5	27.7
3	3	25	94.0	29.6	33	147.8	42.5	25	4052.3	49.3	33	6058.1	68.2
5	3	34	1644.2	98.7	42	3340.7	120.0	34	t.o.	t.o	42	t.o	t.o.
6	4	38	3125.3	296.2	46	5077.0	354.3	-	-	-	-	-	-
6c	4	38	732.1	296.2	46	954.5	354.0	-	-	-	-	-	-
10	3	56	m.o.	m.o.	64	m.o.	m.o.	-	-	-	-	-	-

4.2. Symmetry Reduction with Cadence SMV

If the topology is left unspecified we can take advantage of symmetry reduction as implemented in Cadence SMV and define the types representing node and port indices as scalarsets. Symmetry reduction can not be be applied to configurations with a user-specified topology because assignment of constants to scalarset variables is a symmetry breaking operation. Table 3 gives the results for each configuration without and with symmetry reduction. These experiments were run on a slightly different machine, a Pentium III at 733 MHz. Again, we performed forward model checking but with a reversed variable order. All properties were checked together. We could not handle configurations with 4 or 5 nodes and more than 2 ports and no configuration with 6 nodes within the given memory bounds.

4.3. Bounded model checking with Cadence SMV

Due to the time available to perform the experiments we applied bounded model checking only to selected fully deterministic and fully non-deterministic configurations. Symmetry reduction was not enabled. For the parameter l, the number of runs through the root contention resolution sub-protocol, see Section 3.8, we used values 0 and 2. The resulting path length for the bounded model checker and the user time in seconds are given in Table 4. Apart from the path length k no additional command line parameters had been specified. As SAT solver *zChaff* [ZMM+01] was used. These experiments were run on the same machine as in the previous section.

5. Evaluation

An initial model for specific network configurations in the *SMV* input language can be produced within a couple of days, even with only moderate experience in the subject. Considerably more effort and experience is required to add flexibility to the model while ensuring that verification can be performed within given time and memory bounds.

The input language of *SMV* is well suited to model state machines as used in [1394-1995], [1394-2000]. However, continuous real time constraints cannot be specified and verified. No features are offered by the *SMV* language to model procedural interfaces and advanced data types.

A model in *SMV* should not be much harder to understand than program code written in a low-level procedural language like C. Specifications in temporal logic probably require some explanation. Specification patterns [DAC98] might help here.

Every change of the model requires to rerun the model checker to verify that the specification is still valid. However, for minor changes the parameters of the verification will not have to be modified.

Symmetry reduction is considered most effective in the context of compositional reasoning. In our experiments we relied on fully automatic methods only. Still, symmetry reduction reduced the run time and the memory used in many cases up to 50 %. This allowed us to verify some larger configurations that could not be handled without symmetry reduction within the same memory limit. However, savings in memory seem less stable than savings in time. Though usually fewer memory is required to check each property individually, this option did not enable us to verify larger configurations while in many cases considerably more time was needed. Note, that Yang's version of *SMV* produced much better run times than *Cadence SMV* even with symmetry reduction enabled. Recently bounded model checking has been added to *Cadence SMV*. For our application this new feature did not help to verify larger configurations. As is typical, bounded model checking was faster in finding a counterexample than in (partially) verifying a correct model.

The work most closely related to ours is that of Calder and Miller [CM01]. Their model of communication and their state machine for the explicit state model checker SPIN [Hol91] is more detailed. The type of properties checked are similar. However, they only report data for two fixed topologies with six nodes. In our experiments we verified fixed configurations with up to ten nodes. Given a concrete number of nodes and ports, our setup can handle all topologies in a single run of the model checker.

In principle, our approach is restricted to discrete time and can not handle detailed continuous timing constraints as has been done, for example, by Simons and Stoelinga [SS01]. Therefore, timing related errors as reported by Romijn in [Rom01] would probably remain undetected in our model. In contrast to [SS01] no construction of intermediate models is required for our verification.

6. Conclusions and Outlook

Model checking with *SMV* proved to be very effective for the verification of the Tree Identify Protocol of the IEEE 1394 (FireWire) standard. We were able to model the system rather quickly. The formulation and verification of the requirements was straightforward. Even with finite state model checking generic topologies can be verified. The next step is to move from a fixed number of nodes and ports to an upper bound for these values. More experimental results and application of our methodology during the design process are required.

References

[Ben01] B. Bentley: Validating the Intel Pentium 4 Microprocessor. In: *Proceedings of the 38th conference on Design automation*, June 18 – 22, 2001, Las Vegas, NV, USA. ACM Press, 2001.

[BBF+01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen: *Systems and Software Verification: Model-Checking Techniques and Tools.* Springer Verlag, 2001.

[BCF01] G. Berry, H. Comon, A. Finkel (eds.): *Proceedings of the 13th International Conference on Computer Aided Verification*, Paris, France, July 18 – 22, 2001. Springer Verlag, 2001.

[BCCFZ99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu: Symbolic Model Checking using SAT procedures instead of BDDs. In: *Proceedings of the 36th Design Automation Conference*, New Orleans, LA, USA, June 21 – 25, 1999.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, Y. Zhu: Symbolic Model Checking without BDDs. In: W. R. Cleaveland (ed.): *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Amsterdam, The Netherlands, March 22 – 28, 1999. Springer Verlag, 1999.

[BLM01] P. Bjesse, T. Leonard, A. Mokkedem: Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. In: [BCF01], pp. 454 – 464.

[Bor97] Arne Borälv: The Industrial Success of Verification Tools Based on Stålmarck's Method. In: [Gru97].

[Br86] R. E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation. In: *IEEE Transactions on Computers* 8 (C-35) 1996, pp. 677 – 691.

[Br91] R. E. Bryant: On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. In: *IEEE Transactions on Computers* 2 (40) 1991, pp. 205 – 213.

[BCM92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang: Symbolic Model Checking: 10^{20} States and Beyond. In: *Information and Computation* 98 (2) 1992, pp. 142 – 170.

[CM01] M. Calder, A. Miller: Using SPIN to Analyse the FireWire Protocol - a Case Study. In: [MRS01], pp. 9 – 13.

[CCW+95] S. Campos, E. Clarke, W. Marrero, M. Minea: Verifying the Performance of the PCI Local Bus using Symbolic Techniques. In: *Proceedings of the International Conference on Computer Design*, Austin, TX, USA, October 2 – 4, 1995.

[CCG+02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella: NuSMV 2: An Open-Source Tool for Symbolic Model Checking. In: *Proceedings of the 14th International Conference on Computer Aided Verification,* Copenhagen, Denmark, July 27 – 31, 2002. Springer Verlag, 2002.

[CE81] E. M. Clarke, E. A. Emerson: Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In: D. Kozen (ed.): *Proceedings of the Workshop on Logics of Programs,* Yorktown Heights, NY, May 1981, pp. 52 – 71. Springer Verlag, 1982.

[CFJ93] E. M. Clarke, T. Filkorn, S. Jha: Exploiting Symmetry In Temporal Logic Model Checking. In: [Cou93], pp. 450 – 462.

[CGH+93] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, L. A. Ness: Verification of the Futurebus+ cache coherence protocol. In: D. Agnew, L. Claesen, R. Camposano (eds.): *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Applications,* Ottawa, Canada, April 26 – 28, 1993. North Holland, 1993.

[CGP99] E. M. Clarke, O. Grumberg, D. A. Peled: *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.

[CFF+01] F Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, M. Y. Vardi: Benefits of Bounded Model Checking at an Industrial Setting. In: [BCF01], pp. 436 – 453.

[Cou93] C. Courcoubetis (ed.): *Proceedings of the 5th Workshop on Computer-Aided Verification,* Heraklion, Greece, June 28 – July 1, 1993. Springer Verlag, 1993.

[DAC98] M. B. Dwyer, G. S. Avrunin, J. C. Corbett: Property Specification Patterns for Finite-State Verification. In: M. Ardis (ed.): *Proceedings of the Second Workshop on Formal Methods in Software Practice,* Clearwater Beach, FL, USA, March 4 – 5, 1998. ACM Press, 1998.

[DDH+92] D. L. Dill, A. J. Drexler, A. J. Hu, C. H. Yang: Protocol Verification as a Hardware Design Aid. In: *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors,* IEEE Computer Society, pp. 522 – 525.

[Eme90] E. A. Emerson: Temporal and Modal Logic. In: J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics.* Elsevier and MIT Press, 1990.

[ES93] E. A. Emerson, A. P. Sistla: Symmetry and Model Checking. In: [Cou93], pp. 463 – 478.

[ES00] E. A. Emerson, A. P. Sistla (eds.): *Proceedings of the 12th International Conference on Computer Aided Verification,* Chicago, IL, USA, July 15 – 19, 2000. Springer Verlag, 2000.

[Gru97] O. Grumberg (ed.): *Proceedings of the 9th International Conference on Computer Aided Verification,* Haifa, Israel, June 22 – 25, 1997. Springer Verlag, 1997.

[Hol91] G. J. Holzmann: *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

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

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

[ID96] C. N. Ip, D. L. Dill: Better Verification Through Symmetry. In: *Formal Methods in System Design* 1/2 (9) 1996, pp. 41 – 75.

[KMM00] M. Kaufmann, P. Manolios, J. S. Moore: *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[MRS01] S. Maharaj, J. Romijn, C. Shankland (eds.): *Proceedings of the International Workshop on Application of Formal Methods to IEEE 1394 Standard,* Berlin, Germany, March 13, 2001. Department of Computing Science and Mathematics, University of Stirling, March 2001.

[MRS02] S. Maharaj, J. Romijn, C. Shankland: The IEEE 1394 Tree Identify Protocol. *In this issue*.

[McM93] K. L. McMillan: *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[QS82] J. P. Quielle, J. Sifakis: Specification and Verification of Concurrent Systems in CESAR. In: *Proceedings of the Fifth International Symposium on Programming,* Turin, Italy, April 6 – 8, 1982. Springer Verlag, 1982.

[Rom01] J. Romijn: False loop detection in the IEEE 1394 Tree Identify Phase. In: [MRS01], pp. 25 – 28.

[Sch01] J. M. Schumann: *Automated Theorem Proving in Software Engineering.* Springer Verlag, 2001.

[SB01] V. Schuppan, A. Biere: A Simple Verification of the Tree Identify Protocol with SMV. In: [MRS01], pp. 31 – 34.

[Sht00] O. Shtrichman: Tuning SAT checkers for Bounded Model Checking. In: [ES00], pp. 480 – 494.

[SS01] D. L. Simons, M. I. A. Stoelinga: Mechanical Verification of the IEEE 1394a Root Contention Protocol using Uppaal2k. In: *International Journal on Software Tools for Technology Transfer* 4 (3) 2001, pp. 469 – 485. Springer Verlag, 2001.

[CSMV] Cadence SMV. Available at `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`.

[SMV] The SMV system. Available at `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[YSMV] B. Yang: SMV 2.4b. Available at `http://www.cs.cmu.edu/~bwolen/software/smv/`.

[TWC01] J. Tretmans, K. Wijbrans, M. Chaudron: Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System: Revisiting Seven Myths of Formal Methods. In: *Formal Methods in System Design* 2 (19) 2001, pp. 195 – 215. Kluwer Academic Publishers, 2001.

[URL] `http://www.inf.ethz.ch/personal/schuppan/firewire`.

[Zha97] H. Zhang: SATO: An efficient propositional prover. In: [Gru97].

[ZMM+01] L. Zhang, C. Madigan, M. Moskewicz, S. Malik: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In: *Proceedings of the 2001 International Conference on Computer-Aided Design,* San Jose, CA, USA, November 4 – 8, 2001, pp. 279 – 285.