A Simple Verification of the Tree Identify Protocol with SMV

Viktor Schuppan and Armin Biere

Computer Systems Institute, ETH Zürich, 8092 Zürich, Switzerland
{Viktor.Schuppan,Armin.Biere}@inf.ethz.ch

1 Motivation

Traditional ways of validating and verifying software include testing/simulation and theorem proving. Testing may start as soon as the first prototype exists. No knowledge of a specialised formalism is required. 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 every potential behavior in a test suite. 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 required.

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 the state space to be searched grows and thus the time needed increases.

Other advantages include that model checking can start once the first prototype of the model and the specification have been finished. 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 as well as in industry. See, for example, [1], [2] and [4].

The Tree Identify Protocol of the IEEE 1394 (FireWire) standard [5], [6], proposed as a case study for the application of formal methods [7], is given as a state machine. It can be translated easily into a corresponding model for a model checker. SMV [8], which we used in our evaluation, is probably one of the most widely used model checkers. This contribution describes on-going research on modeling and verifying the IEEE 1394 Tree Identify Protocol with SMV.

2 Model

Our model of a FireWire system is based on the structure as shown in sections 3.7.3.1.2 and E.3.2 of the official standard [5]. A *configuration* consists of a number of *nodes* with a fixed topology. Each node has three *ports* which are either connected to other ports or are inactive. Separate modules contain the implementations of different configurations, and the definitions of the node and port types.

The implementation of a node is similar to the Tree ID state machine in [6]. A state variable holds the state of each node in the state machine. Our set of states includes all states of the Tree Identify Protocol as described in the standard. State T3 is refined (see below) and state S0 is added as an end state.

However, for low-level communication, IEEE 1394 uses two signals with three different states. We abstract from this low-level line-based encoding scheme and directly use the line states given in Tabels 4-27 and 4-28 in [5]. Not all possible line states are relevant for the protocol. See [5] for details. In our implementation each port has a state variable which takes the line state to be transmitted as its value. The peer port has an alias to that variable.

SMV does not offer constructs to express continuous time constraints. Therefore, we use a counter to model time-out and force-root conditions. For the resolution of root contention in the standard, nodes have to wait different amounts of time. We implement this by making nodes choose paths of length one or two in the state machine. For this purpose, state T3 is split up into 5 sub-states.

For the random choice of the bits in the protocol we have implemented two solutions. One is deterministic and supplies a node with a unique sequence of random bits (i.e., a unique id) upon initialization. In the second solution, 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. The latter is closer to the protocol as described in [5], [6]. Due to technical reasons asynchronous execution is only possible with the first solution. For our results on synchronous execution the second variant is used. In the asynchronous case the first solution is employed and a fairness clause is added to ensure that no process is starving forever.

For a certain topology another parameter remains to be chosen: the force-root flag of each node. We have tried configurations with the force-root flag fixed (*deterministic configurations*, marked *det.*) and with the flag left unspecified (*non-deterministic configurations*).

3 Specification

The two most important properties to determine the success of the protocol are stated in the problem description [7]. We have specified three additional properties to ensure that each node arrives in a well defined, safe state at the end of the protocol. Depending on the particular configuraion, further requirements are added. Finally, we include a time-out clause and a clause specifying known potential problems in our implementation.

A typical requirement looks as follows:

```
-- all nodes are in state S0_start
AF (AG in_state_S0_start | timeout | known_problems)
```

When a model checker detects that a requirement does not hold in a given configuration the user is notified and a counter example is produced. This feature helps to track down and correct errors.

4 Results

The specification has been verified for a number of topologies with synchronous execution. Both deterministic and non-deterministic configurations are used. For each configuration, Table 4 lists the number of reachable states as calculated by SMV, the number of bytes allocated, and the user time. Most configurations are easily verified. The verification of the last configuration has been interrupted after 12 hours without results.

We have also experimented with asynchronous executions. These executions demonstrate the importance of placing an upper bound on signal propagation and processing times, a fact which is also reported in [10]. Details can be found in the full version of our paper.

Configuration	# Reachable states	# Bytes allocated	User time [s]
2 nodes *det.*	515	2490368	1.3
2 nodes	38398	4915200	393.72
3 nodes *det.*	7488	3014646	0.98
3 nodes	1.11349e06	7798784	911.28
5 nodes *det.*	1.00329e06	4456448	14.26
5 nodes	4.0333e08	19070976	7074.7
10 nodes *det.*	2.47714e11	11730944	97.47
10 nodes	–	–	>38200

Table 1. Performance

5 Evaluation

The first author who has only started his PhD in the area of formal methods used this case study to gain experience with the SMV system. He has a degree in computer science and some experience in software engineering but not in model checking.

About two weeks were spent reading general introductory texts on the subject. A dedicated introductory course on using SMV should take less time. The prototype presented above was worked out in about one week. As soon as the first model is finished an incremental prototyping/testing/refining cycle started. The user can experience success rather quickly. However, for bigger configurations turn-around times grow. Producing the model seemed easier than coming up with a correct specification. Experience in formulating the model and the specification and fine-tuning the command-line parameters of the model checker to keep running times small is still to be gained.

The version of SMV used here does not support splitting of a model into separate files, thus making modularization and reuse difficult. To solve this problem, some time was spent to combine SMV with a C pre-processor. Commercial tools [11] are available that support these features out of the box.

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 and formulation and verification of the requirements was straightforward. To verify setups with even more nodes the model needs modified and additional model checking techniques should be tried. We have first results for setups with more than 20 nodes. In further experiments we want to investigate how generic setups can be handeled with model checking. We also plan to apply other model checkers to this problem.

References

1. J. Cuellar, D. Barnard, M. Huber: A Solution Relying on the Model Checking of Boolean Transition Systems. In [3], pp. 213 – 251
2. H. Hungar: Specification and Verification Using a Visual Formalism on Top of Temporal Logic. In [3], pp. 305 – 339.
3. M. Broy, S. Merz, K. Spies (eds.): Formal Systems Specification: The RPC-Memory Specification Case Study. Springer Verlag, 1996
4. 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
5. IEEE 1394-1995. Institute of Electrical and Electronics Engineers. IEEE Standard for a High Performance Serial Bus. Std 1394-1995, August 1995
6. IEEE 1394a-2000. Institute of Electrical and Electronics Engineers. IEEE Standard for a High Performance Serial Bus (Supplement). Std 1394a-2000, 2000
7. IEEE 1394 (FireWire) Workshop. Available at http://www.cs.stir.ac.uk/firewire-workshop/
8. K. L. McMillan: The SMV System. November 2000
9. 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
10. D. P. L. Simons, M. I. A. Stoelinga: Mechanical Verification of the IEEE 1394a Root Contention Protocol using Uppaal2k. CSI report CSI-R0009, Computing Science Department, University of Nijmegen, Nijmegen, 2000
11. Cadence SMV. Available at http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/