Explicit State Model Checking in the Development Process for Interlocking Software Systems – Extended Abstract –

Peter Biechele
beXtec GmbH
Kaiserstuhlstr. 3
D-79312 Emmendingen, Germany
Peter.Biechele@beXtec.de

Stefan Leue
tele Research Group, University of Freiburg
Georges-Köhler-Allee 51
D-79110 Freiburg, Germany
leue@informatik.uni-freiburg.de

We report on an ongoing project[1] that addresses the use of explicit state model checking technology in the design of railroad interlocking systems. We discuss our modeling approach, the requirements on the use of formal methods as specified in the pertinent CENELEC standards, and the use of explicit state model checking in requirements verification and test case generation. In the context of test case generation we also illustrate the use of heuristic search strategies in model checking.

I. Case Study

We consider the design of a generic *Track Segment Occupancy and Signaling Control* (TSOSC) software system. Its task is to control the signaling for a track segment in response to an axle counter system that determines the physical occupancy of a track segment. In our abstract requirements model the system obtains as input the axle count as provided by the *Track Segment Occupancy Monitoring* (TSOM) system, reset signals provided by a local operator, and reset signals originating from external operators.

Internally, the system consists of 4 concurrent state machines as shown in Figure 1. The *Err* automaton, which we depict in Figure 2, is responsible for handling internal failure situations. Its state *OK* indicates the absence of a failure. The Res automaton controls the reset mechanism. The Occ automaton represents the physical state of the track segment as per the input from TSOM. The *Relay Control Unit* (RCU) automaton determines the signalling for the segment (states *green* or *red*) based on the state of the other three automata. In the model, the automata synchronize via message passing with the environment, and via shared variables with each other.

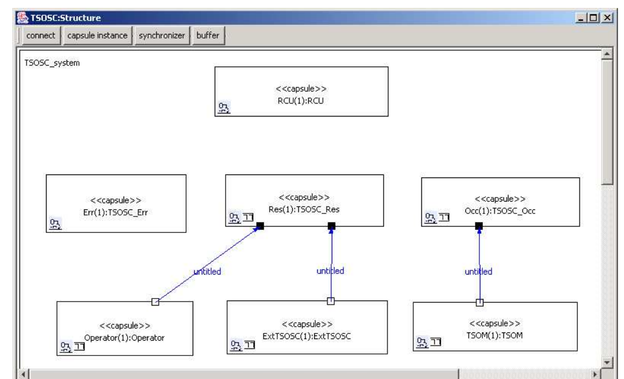

Fig. 1. **Track Section Occupancy and Signaling Control System (TSOSC).**

The capsules Operator, ExtTSOSC and TSOM form the system environment. They represent a human operator, trains passing the axle counter and the remaining TSOSC systems, which we have abstracted away. The typical code size for an implemented TSOSC system lies in the order of 30,000 lines of source code.

II. Modeling Technique

With the help of the VIP editing tool, we produced an abstract requirements model for TSOSC. The VIP Tool implements a dialect of the UML RT notation [1] and converts it to a model specification using the v-Promela language [2], the input language for the SPIN [3] model checker. Figure 1 shows a snapshot of the VIP Tool.

[1] An extended version of this paper is available under the title *Software Quality Assurance for Interlocking Systems using Explicit-State Model Checking Technology* as report tele-2003-01, *tele* Reserach Group, University of Freiburg, Germany, at URL http://tele.informatik.uni-freiburg.de/reports/tele-2003-01.pdf

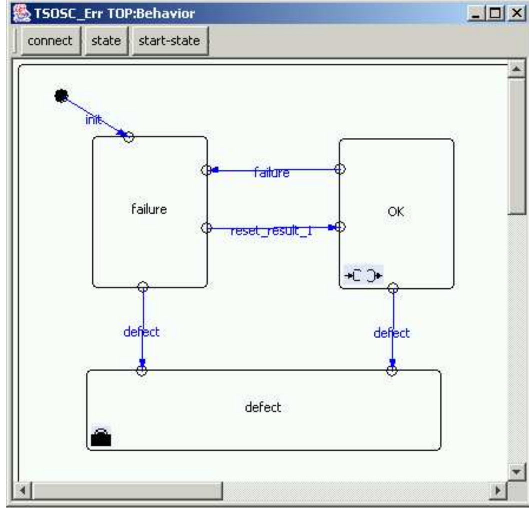

Fig. 2. **Err automaton of the TSOSC.**

III. STANDARDS

In Europe, the development of safety-critical software in the railway interlocking domain is subject to the CENELEC standard EN 50128:2001 [4]. It highly recommends (but does not require) the use of formal, semi-formal and structured methods such as CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM, Z, B and SDL. In terms of verification procedures, the standard highly recommends, amongst others, formal proofs, finite state machine analysis, and formal methods based inconsistency and correctness analysis. Even though state machine analysis and temporal logic are mentioned, it is surprising to see that model checking [5] is not one of the recommended verification techniques.

While finite state machine analysis attempts to reveal inconsistencies such as unspecified receptions in state machines, model checking is capable of verifying more complex temporal properties. It is important to note that model checking not only enforces formal specification of the requirements and software model, but can even show that the requirements are captured by the model. So it is definitely a viable method to fulfill all the requirements of the CENELEC standards.

IV. EXPLICIT STATE MODEL CHECKING

We propose the use of explicit state model checking in the design process of the class of systems we described above. These systems are highly concurrent and mostly asynchronous, which are characterisitcs for which explicit state model checkers are known to work well. Explicit state model checking as a verification technology is becoming increasingly popular in software engineering since it is an automated technique that does not require user interaction, and since property violations are accompanied by counter-examples. The particular model checking tool that we use in our experiments is SPIN (see section II). SPIN is especially well suited in our setting since it provides support for the efficient analysis of asynchronously communicating systems through its use of partial order reduction techniques [5]. The over 10 years of open source availability of SPIN and the well maintained nature of the its code make it well suited to be used in the development of safety critical systems based on the CENELEC standards.

V. PROPERTY VERIFICATION

While property verification using model checking can be useful at various stages of the software design cycle we first illustrate its use to verify requirements models against abstract, high-level requirements. While our requirements model is the Promela model described above, we use Linear Time Temporal Logic (LTL) to fomally capture some informally specified abstract requirements. For instance, it would be intriguing to show that *always, when the error automaton in our requirements model is not in the OK state, then the RCU cannot be in green*, i.e., in LTL:

$$\Box(\neg Err@OK \rightarrow \neg RCU@green).$$

However, due to the asynchronous nature of the state machines this property is not satisfied, as SPIN will prove.

We will be able, however, to prove a stability property such as

$$\Box(((\Box\neg Err@OK) \wedge (RCU@green) \wedge \\ \wedge(\Diamond\neg RCU@green)) \\ \rightarrow (RCU@green)\,\mathcal{U}\,(\Box\neg RCU@green)$$

which states that *always if Err is not at OK, the RCU is at green and the RCU will eventually not be at green, then the RCU will be at green only until it will eventually be forever not at green*.

The stable `RCU` behaviour in case of defect is another important property, which can easily be proven using the model checker. This requirement states that when the error automaton detects a fault that it cannot recover from, i.e., when the `Err` automaton enters the `defect` state, then the `RCU` should never become idle again. More precisely, *whenever the* `Err` *automaton is in an irrecoverable defect state and at the same time the* `RCU` *is not in the* `idle` *state, then the* `RCU` *will never reach the idle state again*:

$$\Box(((Err@defect) \wedge (\neg RCU@idle)) \\ \rightarrow (\Box\neg RCU@idle)).$$

The TSOSC model that we analyzed does actually satisfy both requirements.

VI. TEST CASE GENERATION

The second use of model checking technology that we illustrate is the generation of test vectors based on the state machine model that we sketched above. From an abstract viewpoint, let s characterize a reachable state of the system and g a state reachable from s, then the model checker will disprove the claim $\Box(s \rightarrow \neg\Diamond g)$ by producing an trail from s to g. The events along this trail can be interpreted as a test case. The purpose of this test case is to prove that the system

implementation can be triggered so that it reaches state g from state s.

To concretize this idea, we may wish to test the successful operation of the operator reset mechanism. This means that *from the point where the Operator has pressed the reset button the* `RCU` *will eventually signal that the track section is green*, which can be expressed by the following LTL property:

$$\Box(Operator@pressed \rightarrow \neg\Diamond RCU@green).$$

SPIN produces a trail of 532 steps as counterexample to this formula. Manual inspection shows that this trail does not correspond to an optimally short path from the initial to the final state.

Figure 3 shows a sequence diagram of the 532 steps counterexample. As can be seen in the diagram a reset scenario in this abstract model consists of an internal reset signal given by the operator and an external reset signal, emitted by the external axle counters which have been abstracted away by the `ExtTSOSC` automaton. The `TSOSC Res` automaton notifies the error automaton about the successful reset, which in turn switches to the OK state. But instead of a transition to green in the `RCU` automaton, the `TSOSC Err` automaton switches again into the error state, indicating an internal error. Note that while this is a relatively unlikely sequence of events, the model checker will explore all possible alternatives. At this point the counterexample starts looping the scenario, because for the `TSOSC Err` automaton to become OK, we need a reset scenario again. The trail stays in this loop until at some point the `RCU` automaton is invoked, before the internal error transition in the `TSOSC Err` automaton takes place.

For a test to be carried out, we have to stimulate the software system with the reset signals of the operator and the signals of the external axle counters. Then we just have to wait for the system to report an idle (green) track segment. If the system does not report an idle track segment, the test fails and we have found a bug in the software-system. Obviously, the loop contained in the automatically produced error trace is redundant. For a test we only need to stimulate the software system by one internal and one external reset signal to obtain an idle track segment.

The excessive length of the generated test can be explained by the fact that SPIN uses a simple depth-first-search (DFS) algorithm to traverse the state space in its search for an error state. While DFS is very memory efficient, and therefore frequently used in explicit state model checkers, it does not necessarily find optimally short paths to an error state. Even applying an improved nested DFS algorithm that takes advantage of the structure of the never claim corresponding to the LTL formula resulted in a 117 step long error trail. This is still far from optimal, as manual inspection showed.

The finding of shorter error trails has been addressed by the concept of directed explicit state model checking [6] which employs directed search algorithms such as A* in the state space traversal. A* is guided by heuristic estimates of the minimal number of steps needed to reach a target state and produce optimal or close to optimal error trails. If the heuristic estimate is a true lower bound of the actual distance to an error state, optimally short trails will be found. We say that in this case the heuristics is admissible. As we shall see, inadmissible heuristics do not necessarily mean that improvements cannot be observed, it merely means that there is no guarantee of finding an optimal solution.

We applied this approach using the inadmissible Hamming distance heuristic estimate to the goal state together with the trail improvement technique described in [7]. The Hamming distance estimate takes the number of bit differences between the current state vector and the state vector of the goal state as a measure for the distance to the target state. Obviously, this heuristics in inadmissible since a transition step in the model may flip more than one bit in the state vector which means that the goal state may be reached much faster than estimated.

The experiment generated a test case involving 63 steps. This is a sub-optimal result since the optimal result, which we determined by using a breadth-first search, requires 57 steps. The admissible local state machine distance metrics, finally, provides us with an even shorter test case that consists of only 57 steps. This metrics takes the distance of the current state of the "never claim" automaton in the Promela model as an estimate. The sequence diagram of this trail is shown in figure 4.

The obtained trail is optimal and does not contain the loop anymore. Test case pass and fail criteria are the same as for the long trail.

VII. Conclusion and Related Work

We have demonstrated that temporal logic based requirements capture and explicit state model checking can be successfully applied in the design process for interlocking systems. Their use can help in verifying requirements models, design models and code against requirements, and the underlying state traversal technology can be used to automatically generate test cases. We have shown how to reduce test case length using heurisitc search in the model checking step.

We wish to apply the verification approach at other stages in the life cycle. For instance, it would be desirable to analyze code directly, as it has been proposed in the telecommunications domain [8]. We are also considering the use of real-time models so as to transform the first property mentioned above into a bounded response property.

To improve automated test case generation we will investigate tree-like model checking algorithms since tests are often not just linear sequences, as the current paper assumes. If there are internal decisions over which the test environment has no control, then the test takes the shape of a tree. It will be our goal to adjust the directed model checking approach to this variant. We are also interested in developing heuristics that not only optimize test case length, but also maximize coverage of the Promela model. As a negative effect of obtaining short test cases, the coverage of the model is reduced. Nonetheless, we believe that "short is beautiful", i.e., that an adequately chosen number of short test cases is preferable over a slightly smaller number of very long test cases. For highly concurrent models,

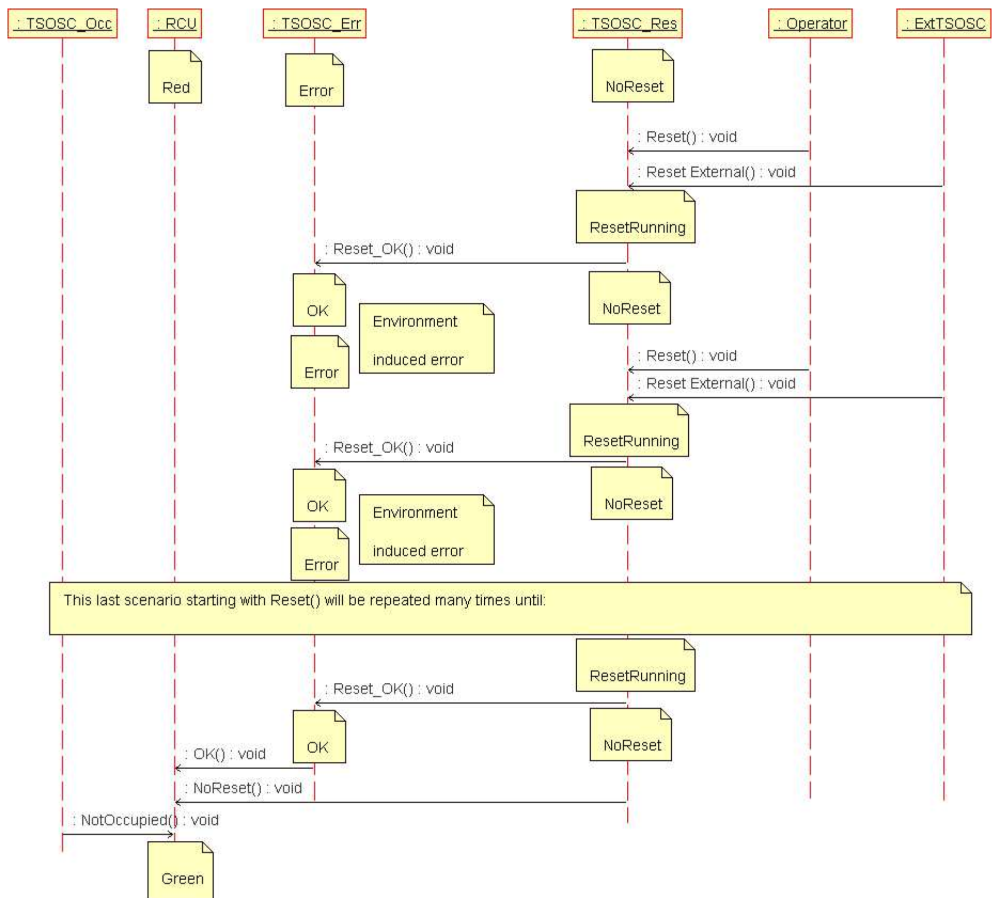

Fig. 3. **Sequence Diagram showing the 532 steps trail.**

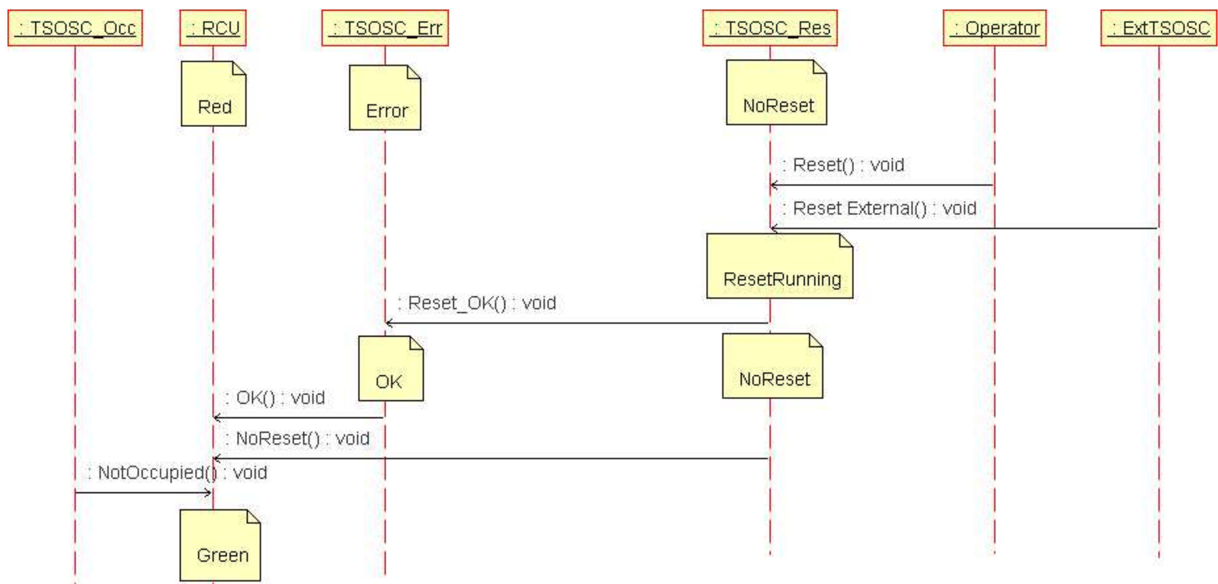

Fig. 4. **Sequence Diagram showing the 57 steps trail.**

which contain lots of internal nondeterminism, we contend that the use of directed model checking technology in test case generation is inevitable in order to obtaine useable test cases. We are finally interested in incorporating probabilistic information into the models which will help in focussing more on likely scenarios rather than unlikely ones.

Our work has many similarities with that described in [9], however, we favour an explicit state model checking approach and the future use of more lightweight scenario-based requirements notations than LSCs. We contend that the use of an LTL notation to capture requirements, as shown in this abstract, is unacceptable in the end-user domain. We will investigate the use of lightweight timeline notations [10] and natural language interfaces to capture temporal requirements [11], [12].

REFERENCES

[1] M. Kamel and S. Leue, "VIP: A visual Editor and Compiler for v-Promela," in *Proc. of the 6th Intl. Conf. on Tools and Algorithms for the Analysis and Construction of Systems TACAS2000*, ser. Lecture Notes in Computer Science. Springer Verlag, 2000, pp. 471–486.

[2] G. Holzmann and S. Leue, "v-Promela: A Visual, Object-Oriented Language for Spin," in *Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE Computer Society, 1999, pp. 14–23.

[3] G. Holzmann, *The Spin Model Checker, Primer and Reference Manual*. Reading, Massachusetts: Addison-Wesley, 2003.

[4] European Committee for Electrotechnical Standardization (CENELEC), "EN 50128:2001: Railway Applications - Communications, signalling and processing systems - Software for Railway Control and Protection Systems," Published in German as: DIN EN 50128 (VDE 0831 Teil 128), Bahnanwendungen - Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme - Software für Eisenbahnsteuerungs- und Überwachungssysteme, VDE Verlag, 2001.

[5] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. The MIT Press, 1999.

[6] S. Edelkamp, S. Leue, and A. Lluch-Lafuente, "Directed Explicit-State Model Checking in the Validation of Communication Protocols," *Software Tools for Technology Transfer*, 2003.

[7] S. Edelkamp, A. Lluch-Lafuente, and S. Leue, "Trail-Directed Model Checking," in *Proc. of the Workshop on Software Model Checking*, ser. Electrical Notes in Theoretical Computer Science. Elsevier, July 2001.

[8] G. Holzmann and M. H. Smith, "Software Model Checking - Extracting Verification Models from Source Code," Kluwer Academic Publ., Oct. 1999, pp. 481–497, also in: Software Testing, Verification and Reliability, Vol. 11, No. 2, June 2001, pp. 65-79.

[9] J. Bohn, W. Damm, H. Witteke, J. Klose, and A. Moik, "Modeling and validating train system applications using statemate and live sequence charts," in *Integrated Design and Process Technology, IDPT-2002*. Societey for Design and Process Science, 2002.

[10] M. Smith, G. Holzmann, and K. Etessami, "Events and Constraints, a graphical editor for capturing logic properties of programs," in *Proc. 5th International Symposium on Requirements Engineering*, Toronto, Canada, Aug. 2001, pp. 14–22.

[11] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini, "Assisting Requirement Formalization by Means of Natural Language Translation," *Formal Methods in System Design*, vol. 4, no. 3, pp. 243–263, 1994.

[12] A. Holt and E. Klein, "A semantically-derived Subset of English for Hardware Verification," in *37th Annual Meeting of the Association for Computational Linguistics: Proceedings of the Conference: 20-26 June 1999*. Association for Computational Linguistics, 1999, pp. 451–456.