Formal Methods in Development and Testing of Safety-Critical Systems: Railway Interlocking System

Tomáš Hlavatý, Libor Přeučil, Petr Štěpán

The Gerstner Laboratory for Intelligent Decision Making
Department of Cybernetics
Faculty of Electrical Engineering
Czech Technical University
Technicka 2, 166 27 Prague 6
Czech Republic

{hlavaty,preucil,stepan}@labe.felk.cvut.cz

Štěpán Klapka

AZD Prague Ltd.
Zirovnicka 2/3146, 106 17 Prague 10
Czech Republic

klapka.stepan@azd.cz

Abstract
The contribution addresses the application of formal methods in functional specification, design and verification of real-time software systems in safety-critical applications. We present basic principles of software verification methods directed towards automatic proof of safety properties against the model of the system. Verification of the railway interlocking system developed by the AZD Prague Ltd. is presented. We discuss advantages and drawbacks of the presented methods and the possibility of using model-checking algorithms in the testing stage of the system development.

1 Introduction

Hard requirements on the precise specification and substantial testing have to be fulfilled during the development of safety-critical systems. Formal specification allows both simulation and automated generation of target code as automatic proof of its correctness and consistency. The formal model of the system can also be used as an oraculum for the generation of test sequences. Although the development of the formal specification is expensive, its cost is partially compensated by the elimination of mistakes in the system specification and implementation stages. Formal specification itself cannot replace the testing of the system. Therefore testing is still an important and complementary activity which can be automated on the basis of the formal specification.

Safety-critical systems are often modeled by reactive systems. These systems are usually engaged in applications with high reliability and safety requirements thus the need of precise formal specification arises. The model of a system is often written in a language with well-defined syntax and semantics with underlying mathematical background. Some properties of the system can be expressed in a logic. Opposite to theorem proving, model-checking used in this paper can be completely automatic.

We present a railway interlocking system as a practical example of verification using model-checking. Verification of its safety properties was used as a basis for validation purposes by third-party validators. We briefly present an architecture of the system and discuss only few safety properties of a small automaton which controls moving of trains on tracks.

Model-checking algorithms are recently used in several practical tasks. Efficient planners are designed on the basis of counterexample generation algorithm. The task of test sequence generation can be viewed as a planning problem. Moreover, conformant planners with nondeterminism in an initial state and actions allows a test sequence generation for distributed systems which are often loaded by nondeterminism that cannot be controlled.

First, the description of reactive systems is given in sec. 2. Next, model-checking algorithms are introduced in sec. 3 and the industrial railway interlocking system is sketched in sec. 4 of this contribution. Finally, we discuss the automatic generation of test sequences based on the algorithm for generating counterexamples in model-checking (sec. 6).

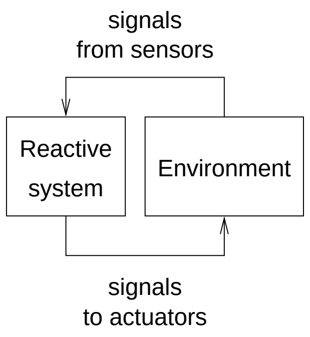
2 Reactive Systems

Basically, software systems can be divided into three classes [4, 18]: transformational, interactive, and reactive systems. In this paper, we are mainly interested in reactive systems. But first, we give a brief description of the first two kinds of systems.

Transformational systems terminate with the result computed from an input data (e.g. compilers). *Interactive systems* communicate with their environment at their own speed (e.g. concurrent processes in operating systems, web servers, database servers). They are able to synchronize with their environment, i.e. making it waiting.

Compared to the above systems, *reactive systems* react to an environment that cannot wait (e.g. control systems of physical processes). In contrast with most interactive systems, they are generally intended to be *deterministic* both in the term of input/output behavior and time response. Their description involves *concurrency* for several reasons: i) They run in parallel with their environment. ii) They are often implemented on distributed architectures for reasons of speed, fault-tolerance, or physical distribution requirements. iii) It is often convenient to describe them as sets of concurrent processes. They are submitted to critical reliability requirements. The most critical systems either are reactive, or contain reactive parts.

There are two basic ways how to implement a reactive system fig. 1. In *event-driven/asynchronous* scheme, each reaction is triggered by an input event. *Sampling/synchronous* scheme consists in periodically sampling the inputs. Both cases are typically implemented as an *automaton*: the states are the valuations of the memory, and each reaction corresponds to a transition of the automaton. The system is synchronous if such a transition is considered atomic (i.e. input changes are only taken into account between two reactions) and such a reaction takes no time. An atomic reaction is called an instant (logical time), and all the events occurring during such a reaction are considered simultaneous.



a) Parallel composition of the reactive system and its environment

```
initialize memory
for each event do
    compute outputs
    update memory
end
```

b) Event driven scheme (asynchronous control)

```
initialize memory
for each period do
    read inputs
    compute outputs
    update memory
end
```

c) Sampling scheme (synchronous control)

Figure 1: Execution schemes of reactive systems

When introducing concurrency, the advantage of synchronous systems lies in their modular composition. In the event-driven scheme, some automata may stay idle, when not triggered by events coming either from the environment or from other automata. In the sampling scheme, when automata are composed in parallel, a transition of their product is made of simultaneous transitions of all of them. When participating in such a compound transition, each automaton considers the outputs of others as being part of its own inputs. This instantaneous communication is called *synchronous broadcast*. Compared to asynchronous concurrency, the synchronous product can preserve determinism, a highly desirable feature in reactive systems design.

There are two major fields where the synchronous model is used: i) in synchronous circuit design, where the system is modeled by communicating Mealy machines, and ii) in control engineering, where the system is modeled by differential or finite-difference equations, block diagrams, etc.

The reactive systems often require precise formal specification and verification of their correct design. Several formal description methods for reactive system were developed. The basic approach to verification is theorem proving where the system is described in a logic and the proof of its properties is made by a deduction system. Deduction systems are correct and complete only for a small class of logics, e.g. reasoning about natural numbers cannot be done automatically in principle. The need of human interaction with a deduction system increase the cost of the system development. In some cases, it is sufficient to use simple models with finite and/or binary variables, simple constraint and properties. Reactive systems usually fall in this case. A different approach to theorem proving was developed in order to automatically reason about systems. It is based on the state space exploration.

3 Symbolic Model-Checking

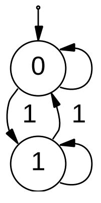
Model-checking [12, 25] has been successfully applied to a wide variety of practical problems, including hardware design, protocol analysis, operating systems, reactive systems analysis, fault tolerance, and security. The chief advantage of model-checking over the competing approach of theorem proving is *complete automation*. Whereas human interaction is generally required to prove all but the simplest theorems, model checkers can explore the state spaces for finite, yet realistic, problems without human guidance.

A model-checking specification consists of two parts. One part is a state machine defined in terms of variables, initial values for the variables, environmental assumptions, and a description of the conditions under which variables may change value. The other part is temporal logic constraints over states and execution paths. Conceptually, a model checker visits all reachable states and verifies that the temporal logic properties are satisfied over each possible path. Model checkers exploit clever ways of avoiding brute force exploration of the state space [24, 25, 34]. If a property is not satisfied, the model checker attempts to generate a counterexample in the form of a trace or sequence of states. For some temporal logic properties, if the property states that at least one possible execution path leads to a certain state and in fact no path leads to that state, there is no counterexample to exhibit.

There are two main approaches to the model-checking depending on the formal representation of the model. The first one is based on the *explicit* representation of the model which is a finite graph. The model-checking procedure then relies on the usual algorithms from the graph theory [24]. Another approach is based on *symbolic* representation of the model. In this case, the model-checking procedure is specified in terms of symbolic manipulations with terms in propositional logic [25]. Efficient algorithms exist that can handle large systems [34]. They are often based on special representation of a propositional logic formula, e.g. Boolean Decision Diagrams (BDDs), Binary Expression Diagrams (BEDs), etc.

3.1 Representation of Finite State Machines

A finite state machines (FSM) can be represented *explicitly* (e.g. by table or graph) or *symbolically* (e.g. by a logic expression) as shown on fig. 2. As the size of a FSM grows, the problem of *state explosion* arises. However, symbolic representation can handle this problem. Several possible symbolic representations were discovered by researches in the field of model-checking [25, 34].



a) Graph

initial state	current state	action	next state
$\rightarrow$	0	0	0
$\rightarrow$	0	1	1
	1	0	1
	1	1	0

b) Table

$$\begin{aligned} I &= \neg s \\ T &= (\neg s \wedge \neg a \wedge \neg s') \\ &\vee \ (\neg s \wedge a \wedge s') \\ &\vee \ (s \wedge \neg a \wedge s') \\ &\vee \ (s \wedge a \wedge \neg s') \end{aligned}$$

c) Expression

Figure 2: Representation of finite state machines by graph, table, and expression

The system is usually symbolically represented as a finite Kripke structure, or modified finite Kripke structure [34]. The transition function T of a FSM is then represented as a relation on current state, next state, and input variables in the propositional logic, i.e. $T(s, a, s')$. Such a symbolic representation allows efficient state space exploration as well as automatic proof of some temporal properties. When we are not interested in actions, it is sufficient to use only $T(s, s')$ for the transition relation.

3.2 Properties in Temporal Logic

The property is a mathematical or logical statement about the system that we want to verify. A finite system can be symbolically represented in term of formulae in propositional logic (as shown on fig. 2c). There are two basic tasks in propositional logic. The *Satisfiability Problem* is the problem of determining whether a variable assignment exists for the given propositional formula ϕ, such that ϕ evaluates to true for this assignment. The *Tautology Problem* is the problem of determining if ϕ evaluates to true for all possible variable assignments. These two problems are complementary in the sense that $Taut(\phi) \equiv \neg Sat(\neg\phi)$ [15, 17].

Propositional logic can be extended to quantified Boolean formula (QBF) by introducing the existential quantifier $\exists$ defined as

$$\exists x : \phi \equiv \phi[0/x] \vee \phi[1/x], \tag{1}$$

where $\phi[b/x]$ means a substitution of b for x in ϕ. The universal quantifier $\forall$ can be obtained from the existential quantifier using negation:

$$\forall x : \phi \equiv \neg\exists x : \neg\phi \equiv \phi[0/x] \wedge \phi[1/x]. \tag{2}$$

A variable is said to be *free* in formula ϕ if it is not bound by a quantifier. Note that solving the satisfiability/tautology problem for ϕ corresponds to adding existential/universal quantifier for all free variables in ϕ and expanding the resulting QBF to a propositional logic formula using eq. 1 and eq. 2. The resulting propositional logic formula contains no variables and can be reduced to either true or false.

There are many kinds of temporal logics [15]. The most wide-spread temporal logic is *Computation Tree Logic* (CTL) [12] that is used to describe the specification of a finite state machine. Given a state in a Kripke structure M and a CTL specification ϕ for M, then ϕ either holds or does not hold for that state. Thus a CTL formula represents a set of states for which the CTL formula holds. The model-checking is the process of determining whether a Kripke structure M is a model of a CTL formula ϕ (i.e. $M \models \phi$).

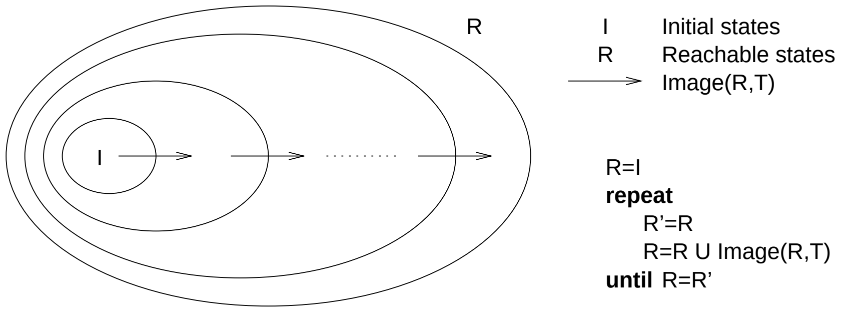
A CTL formula contains a propositional logic part. Furthermore, it also contains a number of temporal operators each consisting of a path quantifier and a path operator [12]. There are two path quantifiers: **E** ("there exists an infinite path") and **A** ("for all infinite paths"). There are five path operators: **X** ("next state"), **G** ("globally"), **F** ("future"), **U** ("until"), and **R** ("release"). The semantics of a CTL formula is a set of states. Some CTL formulae can be evaluated directly, while some are computed using fixed-point iteration transforming sets of states to another sets of states until a fixed-point is reached. The fixed-point iteration is the monotonic sequence of sets of states $Q_0, Q_1, \ldots$ where either $Q_i \subseteq Q_{i+1}$ or $Q_i \supseteq Q_{i+1}$ for all $i \geq 0$.

Some common CTL formulae are: **EF** ϕ ("It is possible to reach a state where ϕ holds"), **AG** ϕ ("ϕ is an invariant" or "The system will never get to the bad state $\neg\phi$"), **AG** $(Req \rightarrow \mathbf{AF}\ Ack)$ ("All requests will eventually be acknowledged"), etc.

3.3 Automatic Proof and Counterexamples

Model-checking algorithms are based on the *reachability analysis* [1, 7] exploring states reachable from an initial state. In symbolic model-checking, the set of reachable states R is computed by the fixed-point iteration (see fig. 3). The algorithm always terminates because the reachable state space is finite and the sequence is monotonic. The function $Image(s, T)$ returns states reachable from states s in one step where T is the transition function of the system. It is computed as

$$Image(s, T) = (\exists s'.T(s, s'))[s'/s]. \tag{3}$$



```
R=I
repeat
    R'=R
    R=R U Image(R,T)
until R=R'
```

Figure 3: Computing the set of reachable states by the fixed-point iteration

Because the property to be verified is also translated to the set of states where it holds, its proof consists in the comparison of the reachable states and states satisfying the property. E.g. the property **AG** p says that p holds in every reachable state. It means that $R \Rightarrow p$ in propositional logic. If it is tautology then the property is satisfied else the contradiction was found. If the contradiction results in the nonempty set of states then a counterexample can be generated. The basic algorithm for finding the counterexample is sketched on fig. 4. First, preimages (i.e. states s_{pre} from those the given states s are reachable) are computed as

$$Preimage(s, T) = \exists s_{pre}.(T(s_{pre}, s)[s/s']). \tag{4}$$

until an initial state was reached. Next, the sequence of images is computed and at each step, one state from the computed set of states is selected. The computation stops when a goal state is found. The computed path from the initial state to the goal state is the counterexample for the given property.

It is not necessary to describe properties in temporal logics. An automaton which indicate whether the property is false or true can be translated from the temporal logic expression or directly specified in the formal language used for the system specification. Such an automaton is called *observer* [20, 19] or *monitor* [2] and is put in parallel with the system. In the verification of synchronous systems it is called *synchronous observer* (see fig. 5). The observer returns true (or OK) if the property is satisfied else it returns false (or fail). The proof then consists in showing that the observer always returns true. It can be expressed by the CTL formula as **AG** ok, i.e. ok is invariant. The advantage of observers consists in the possibility of describing more complicated properties than can be expressed in CTL using the formal language used for the definition of the system.

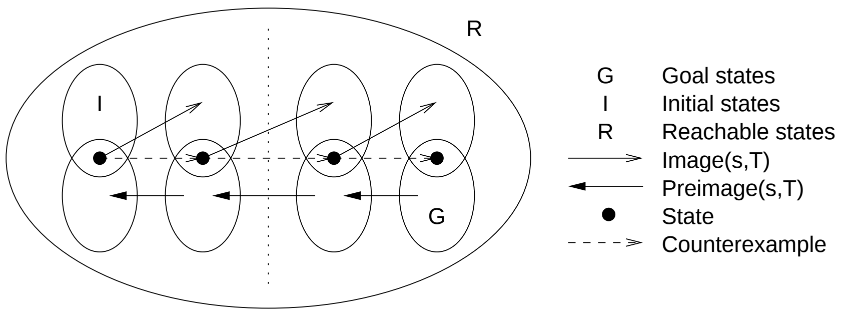


Figure 4: Generation of counterexamples

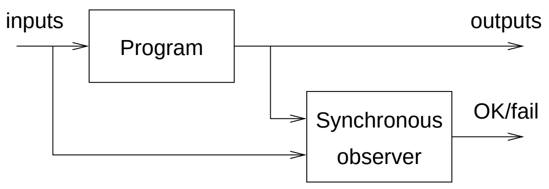


Figure 5: Synchronous observer

3.4 Data Structures in Model-Checking

The proof of properties leads to the tautology checking so the final result is the expression true or false. Some tasks in model-checking lead to the Satisfiability Problem, the first problem proved to be NP-complete [17].

The time/memory efficiency of the above algorithms is hardly dependent on the representation of the propositional formula. Several representations of propositional formulae can be used. Conjunction Normal Form (CNF) and Disjunction Normal Form (DNF) are often replaced by Binary Decision Diagram (BDD) [3], which is another normal form allowing constant time for both, tautology and satisfiability checking (see fig. 6a). The well known disadvantage of BDDs lie in their exponential memory complexity for some sorts of Boolean functions they represent (fig. 7). Some representations embody compromise in time/memory efficiency by reducing redundancies in the representation of a propositional logic formula, e.g. Boolean Expression Diagram (BED) [34] shown on fig. 6b.

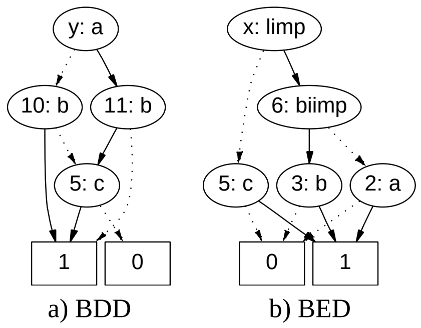


Figure 6: BDD and BED for the formula $(a \oplus b) \vee c$

Recently, new approaches to model-checking and reachability analysis are often based on translations to SAT [6, 1, 7]. They profit from the research on efficient SAT-solvers, such as improvements of Davis-Putman Procedure or Stalmarck Method based on Dilemma Proof [31]. On the other hand, some approaches enhance the expressiveness of the model description language. An example can be Integer Logic Programming (ILP) which is a diverse technique transforming Boolean formulae into integer equations (fig. 8) and solving a problem by Constraint Logic Programming (CLP) [26]. The problem is then formulated as an integer optimization problem allowing integer constraints in the model description.

The main advantage of model-checking lies in the automatic proof which prevent human interventions. In the next section, we describe a practical task of formal verification of the railway interlocking system. First, the system

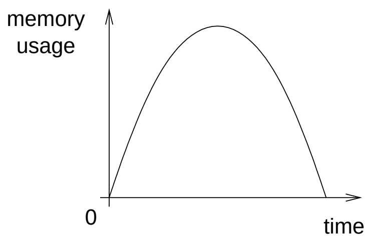


Figure 7: Typical memory requirements during BDD computations

$$\begin{aligned} a \vee b &\equiv a + b \geq 1 \\ a \wedge b &\equiv a + b \geq 2 \\ a \wedge \neg b &\equiv a + (1 - b) \geq 2 \end{aligned}$$

Figure 8: Examples of ILP translations

is described and then some safety properties are mentioned. Finally, we discuss verification of the properties by some model-checking tools.

4 Railway Interlocking System

The railway interlocking system considered in this work is the Line Block (LB) developed by AZD Prague Ltd. [14]. It is responsible for supervision of trains on tracks between railway stations (fig. 9). The architecture of the LB is illustrated by the fig. 10. The Diagnostic System attached by a fast serial line does not directly participate on the control. It only serves as an observer that is responsible for recording of telegrams, on-line diagnostics and monitoring of the LB state.

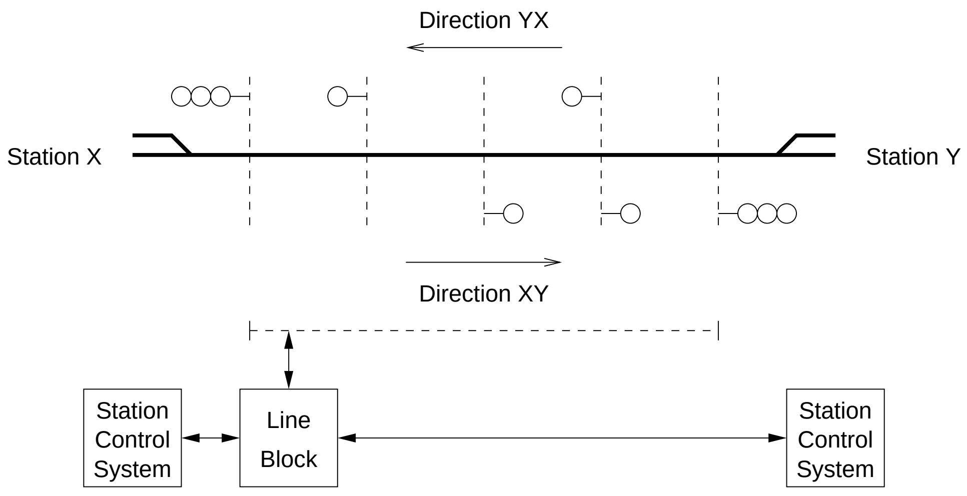


Figure 9: Configuration of the Line Block

The LB follows a centralized concept of the Central Control Unit (CCU) with attached Peripheral Control Units (PCUs). As the CCU is responsible for execution of the overall control strategy of the system, the PCUs are expected to drive particular interlocking system components (e.g. signalling lights) and detect external events (e.g. status of track circuits, train position, etc.). All the units are interconnected through a fast serial line. The communication link and each of the units are backed-up by simultaneous running of different releases of the same program on distinct processors and using physically separated data lines.

Each PCU is allowed to communicate only with the CCU. The CCU itself operates in a loop and executes a finite state machine program called the Control Logic Automaton (CLA). From this point of view, it can be considered that the CCU works in three steps: i) Reads inputs from the other modules; ii) Transforms the inputs

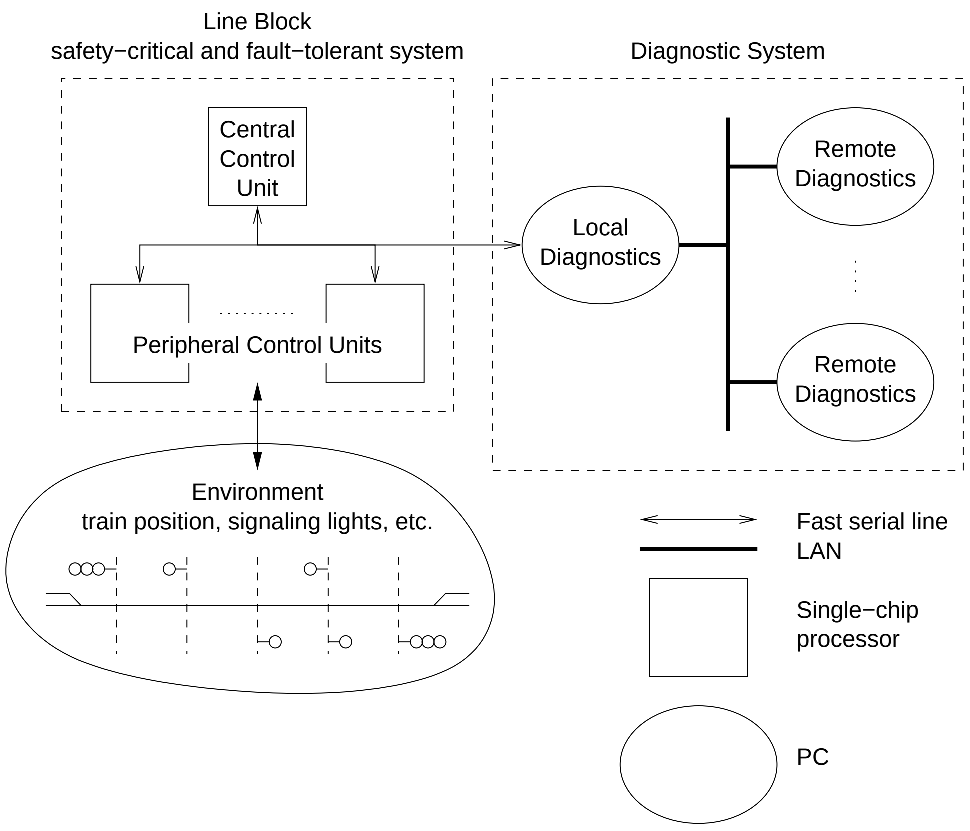


Figure 10: Architecture of the Line Block

to its outputs through the CLA; iii) Sends output values to the other modules. As the CLA executes the main control algorithm of the interlocking system, it can be modeled by a synchronous Mealy automaton. The Line Block Direction Control (LBDC) shown in the fig. 11 is considered for a backbone of the CLA. Therefore, the LBDC has been chosen for the test-case in the following.

5 Verification of LBDC

In this section, we list some of the most meaningful properties verified on LBDC [23, 22]. Their verification is performed using the three model-checking tools based on BDD model-checking algorithms:

- SMV (Symbolic Model Verifier) uses its own language for the model description. Properties can be expressed in temporal logics CTL or LTL (for bounded model-checking) [6]. Both synchronous and asynchronous systems can be verified. Also the system can be nondeterministic.
- Lesar verifies invariants expressed by synchronous observers using the synchronous data-flow language Lustre [19, 20]. The system is modeled by a Lustre program which is an abstract model of a deterministic synchronous Mealy automaton.
- Mocha [2] uses its own language for the model description. It supports compositional and hierarchical verification methodologies. Properties can be written both in CTL and LTL. More complicated properties can be verified by monitors.

Let us mention the major LBDC properties. These can be described by formulae built on variables shown in tab. 1. We list three properties in the data-flow language Lustre and we give also a CTL formulae for the first property.

P1 The LBDC permission can obtain at most one station.

```
p1  = not (X_HST and Y_HST)
```

It can be expressed in CTL as **AG** $\neg(X\_HST \wedge Y\_HST)$

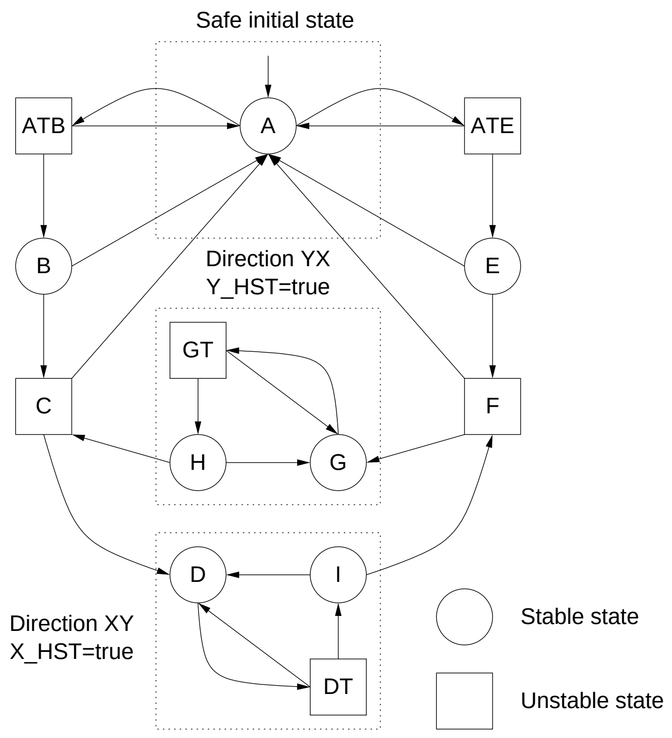


Figure 11: Line Block Direction Control

Variable	Meaning
ST	state of the LBDC
T	counter for unstable states of the LBDC
UVT	track is completely free
X_HST	station X has the LBDC permission
Y_HST	station Y has the LBDC permission

Table 1: Variables of the Line Block Direction Control

P2 Whenever a station loses the LBDC permission, the other station cannot obtain the LBDC permission for next *n* ticks.

```
p2x = not Y_HST -> implies (pre X_HST and not X_HST, hold (n, not Y_HST))
p2y = not X_HST -> implies (pre Y_HST and not Y_HST, hold (n, not X_HST))
p2  = p2x and p2y
```

P3 A station can obtain the LBDC permission only and only if the requested track is free.

```
p3x = not X_HST -> implies (not pre X_HST and X_HST, pre UVT)
p3y = not Y_HST -> implies (not pre Y_HST and Y_HST, pre UVT)
p3  = p3x and p3y
```

We proved correctness of all listed properties automatically in less than 1 second by all the three tools: SMV, Lesar, and Mocha. No contradiction was found because we took LB specification, which has already been widely tested.

Model-checking algorithms are recently engaged in several fields. An example from research on planners can be taken. The competitive state of the art planners are recently based on three approaches. The first one come out from planning graph analysis [8, 33]. It efficiently generates parallel plans but does not allow nondeterminism both in an initial state and actions. The second approach is based on translations of a planning domain to propositional logic. A SAT-solver is then used for plan generation. The third approach is based on model-checking algorithms. The last two approaches allow nondeterminism both in an initial state and actions. Such planners are called conformant [33, 11].

Planners are often used during the testing stage of software development, namely in automated generation of test sequences. We discuss an exploitation of model-checking algorithms during the test sequence generation in the next section.

6 Testing Methods Based on Model-Checking

There is a strong motivation for automatic test sequence generation coming from industry. For control-dominated systems like the railway interlocking system described above, two basic methods for test generation can be used. The first one is based on fault model. The other strategies check FSM equivalence of specification and implementation of the system. Briefly, FSM equivalence checking have to test that all states and transitions are correctly implemented. Test sequences then consist of two parts:

- α-part to verify all states, and
- β-part to verify all transitions.

Most of the methods not only check each transition in an FSM specification at least once, which corresponds to the branch coverage criteria often used in software testing, but also verify the tail state of the transition to obtain high fault coverage and to guarantee conformance in the context of a more general fault model [29]. The tail states are generally verified by the state identification techniques [5, 32, 27, 10, 9]. Without state identification, test cases do not guarantee to detect the fault that the machine enters a different state than specified.

In particular, to achieve a particular test purpose which is a certain transition to be tested, the following steps have to be performed (see fig. 12):

1. Bring the FSM from its initial state to the starting state of the transition under test using the shortest input sequence possible (called a preamble of the test case).
2. Execute the transition and check the observed output.
3. Look for hidden states up to a pre-defined maximum depth.
4. Check a tail state of the transition by observing its reaction to a pre-selected set of state identification sequences, which can verify the correctness of the tail state (a test body to achieve the test purpose).
5. Apply an input sequence to return to the initial state of the FSM (a postamble of the test case). The user may specify a so-called homing sequence which is expected to take the FSM from any state back to the initial state.

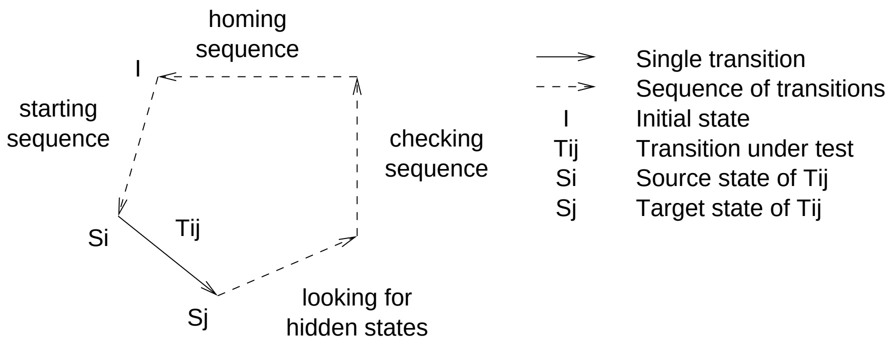


Figure 12: Basic testing algorithm

Several exhaustive methods have been developed such as Transition Tour, W-Method, Wp-Method, Checking Sequences, Generalized Wp-Method, Unique Input Output Sequence, etc. However, these methods are practical only for testing of small systems with few states. For larger systems, exhaustive testing is infeasible. Here, guided testing with human interaction take place in order to increase test suite coverage.

In general, the problem of finding a path from an initial state to a goal state arises. In this case, algorithm for counterexample generation can be used. SCR tool [16] generates test sequences using mutation analysis. After the comparison of the original model and its mutant is performed, a counterexample is generated by a model-checking tool. This counterexample is the optimal path from an initial state to the goal state discovering the fault.

Another approach comes out from synchronous observers used for verification of the system [28, 30, 13]. The property expressed by an observer is translated to an automaton with a failure indicator. The test generation tool then tries to set up the system under test into a state where failure is indicated.

We plan to develop a method for the automated conformance testing of reactive systems based on techniques used in symbolic model-checking [21]. We define some partial testing strategies based on different coverage criteria. In order to automatically select test sequences, evaluate test set coverage and decide whether the test passed or failed, precise formal specification of the system under test is required. Such a specification consists of an initial state of the system and a deterministic transition relation. The set of reachable states is computed using algorithms for reachability analysis used in symbolic model-checking. Any set of states is represented by a Boolean formula over current state, next state, and input variables. Such a symbolic representation allows to avoid explicit state space traversal, reduce memory requirements, and to simplify operations on sets of states, which can be computed as logical operations on Boolean formulae. There is no restriction on the representation of Boolean formulae. However, we can choose between the canonical form called BDD (Binary Decision Diagram) and its generalization called BED (Boolean Expression Diagrams).

The set of reachable states is computed by the fixed-point iteration. For each iteration, we store the set of states R_d added at the corresponding depth d fig. 13. This set is then used for selection of the goal state, which have to be reached by a test sequence. When the set of reachable states is computed, we generate and apply test sequences until the selected coverage criterion is met. Generation of a test sequence is based on the backward fixed-point iteration from a state selected from the set of states added at the depth preferred by the coverage criteria to the initial state (as sketched on fig. 4). Generated path leads from the initial state to the goal state. Test sequence is composed from the path leading from the initial state to the goal state and from the path leading back to the initial state. During the generation of test sequence, states/transitions that have not been covered are preferred. In order to measure test set coverage, we simultaneously build the set of reachable states from states/transitions that have been examined by test sequences. Comparison of the set of reachable states computed from specification and the set of reachable states build by applied test sequences gives us the coverage measure. The coverage criterion has to be modified for the testing of safety properties. Such a criteria has to cover all states/transitions related to the tested property. One must be aware that testing of some safety properties can lead to exhaustive testing.

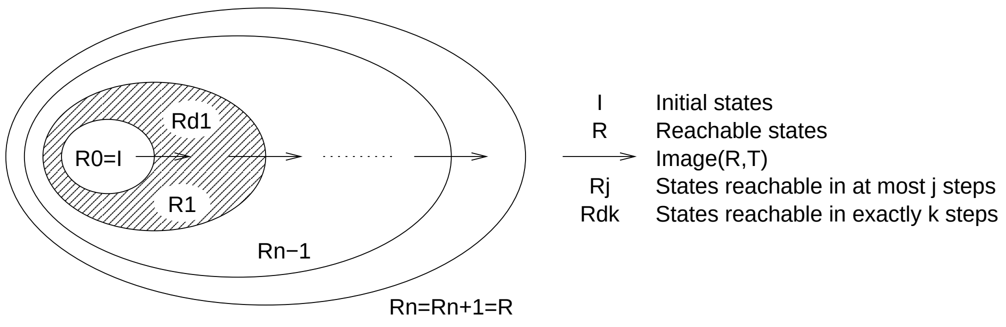

Rd1
R0=I
R1
Rn−1
Rn=Rn+1=R
I Initial states
R Reachable states
Image(R,T)
Rj States reachable in at most j steps
Rdk States reachable in exactly k steps

Figure 13: Reachability analysis for test sequence generation

Our method has some advantages: i) Testing is completely automatic. ii) Several kinds of state of the art model-checking algorithms for symbolic computation can be used. iii) It has a deep formal background.

Of course, we are aware of some drawbacks: i) Testing strategy leads to partial testing, i.e. full fault coverage is not achieved. ii) Only Boolean systems can be tested. iii) Drawbacks of symbolic model-checking algorithms are adopted, e.g. memory requirements growing exponentially depending on the size of the system and the kind of transition relation. However, the rapid progress in the state of the art model-checking algorithms reduces some drawbacks significantly. iv) The need of computing of the set of reachable states which is the most time/memory consuming process.

We plan to extend the method by the possibility of testing nondeterministic systems. It should be possible because state of the art conformant planers are also based on algorithms used in symbolic model-checking. These planners handle nondeterminism in actions as well as a nondeterminism in an initial state. Planning under nondeterminism in actions allows on-the-fly generation of test sequences.

7 Conclusion

The presented work aims to skim through the application of the formal methods for proper software specification, design and verification of safety properties of the railway interlocking system. We performed experiments on the safety-critical railway control system developed by the AZD Prague, Ltd. We wrote formal specification of the system and verified some safety properties using the tools SMV, Lesar, and Mocha. We develop a prototype implementation of the testing tool that allows us effective experimentation with BDD and BED packages.

Although some initial tests of the Central Control Unit algorithms of the railway interlocking system have already been performed, further testing activities are still in progress. As the current implementation of the control system is hand-written in C, the future challenge is to compare behavior of the CCU running the Lustre system generated C code with the current hand-written implementation.

Acknowledgement

This research is supported by the State Department of Education within the frame of the project "Decision Making and Control for Manufacturing" No. MSM 212300013 and by the grant No. CTU 300108313.

References

[1] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on SAT-solvers. In Susanne Graf and Michael Schwartzbach, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems TACAS, Berlin, Germany*, volume 1785. Springer-Verlag, 2000.

[2] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, July 1999.

[3] H. Andersen. Introduction to BDDs, 1997.

[4] Charles Andre. Representation and analysis of reactive behaviors : A synchronous approach. In *CESA'96, IEEE-SMC, Lille, France*, July 8 1996.

[5] B. Beizer. Black-box testing: Techniques for functional testing of software and systems, 1995.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[7] P. Bjesse, M. Sheeran, and G. Stalmarck. SAT-based model checking: A tutorial and overview, 2001.

[8] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[9] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Automatic executable test case generation for extended finite state machine protocols.

[10] R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems, 1998.

[11] Alessandro Cimatti and Marco Roveri. Conformant planning via model checking. In *ECP*, pages 21–34, 1999.

[12] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Symposium on Principles of Programming Languages*, pages 117–126, 1983.

[13] Lydie du Bousquet, Farid Ouabdesselam, Jean-Luc Richier, and N. Zuanon. Lutess: A specification-driven testing environment for synchronous software. In *International Conference on Software Engineering*, pages 267–276, 1999.

[14] A. Faran, J. Houser, and S. Klapka. Logical dependencies in ABE-1. AZD Prague Ltd., Internal document, 2000.

[15] D. Gabbay, C. Hogger, J. Robinson, and D. Nute. Handbook of logic in AI and logic programming, 1994.

[16] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc., Joint 7th Eur. Software Engineering Conf. and 7th ACM SIGSOFT Intern. Symp. on Foundations of Software Eng. (ESEC/FSE99), Toulouse, FR*, Sept. 6–10 1999.

[17] J. Gu, P. Purdom, J. Franco, and B. Wah. Algorithms for the satisfiability.

[18] N. Halbwachs. Synchronous programming of reactive systems. *The Kluwer international series in Engineering and Computer Science*, 1993.

[19] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre. In *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[20] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Workshops in Computing*. Springer Verlag, June 1993.

[21] T. Hlavaty and L. Preucil. Automated testing of reactive systems. In *Workshop CTU 2002*, Prague, Czech Republic, February 2002. Czech Technical University.

[22] T. Hlavaty, L. Preucil, and P. Stepan. Case study: Formal specification and verification of railway interlocking system. In *Euromicro 2001*, ISBN 0-7695-1236-4, pages 258–263, Warsaw, Poland, September 2001. New York: IEEE Computer Society Press.

[23] T. Hlavaty, L. Preucil, and P. Stepan. Formal methods in development and testing of railway interlocking systems. In *International Conference Railway Traction Systems*, pages 173–192, Capri, Italy, May 15–17 2001.

[24] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[25] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[26] Henry A. Kautz and Joachim P. Walser. State-space planning by integer optimization. In *AAAI/IAAI*, pages 526–533, 1999.

[27] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[28] X. Nicollin, P. Raymond, and D. Weber. Automatic testing of reactive programs. In *19th IEEE Real-Time Systems Symposium, Madrid*, December 1998.

[29] A. Petrenko, G. v. Bochmann, and M. Yao. On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems*, 29(1):81–106, 1996.

[30] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *RTSS*, pages 200–209, 1998.

[31] Mary Sheeran and Gunnar Stalmarck. A tutorial on stalmarcks's proof procedure for propositional logic. In *Formal Methods in Computer-Aided Design*, pages 82–99, 1998.

[32] Q. Tan, A. Petrenko, and G. Bochmann. A test generation tool for specifications in the form of state machines, 1996.

[33] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[34] P. F. Williams. *Formal Verification Based on Boolean Expression Diagrams*. PhD thesis, Department of Information Technology, Technical University of Denmark, 2000.