Proving Completeness of Properties in Formal Verification of Counting Heads for Railways

Sebastian Kinder Rolf Drechsler

Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{kinder,drechsle}@informatik.uni-bremen.de

Abstract

The demand for safety of electronic devices is high. Especially in safety-critical systems, e.g. electronic railway interlocking systems, safety is an important issue. Nowadays these systems are tested and simulated with a manually created set of test cases. But testing is a very cost-intensive procedure and can never reach a complete coverage for large designs. Hence, an efficient way to formally verify these systems is required.

In this paper we present the formal verification of Counting Heads for railways, a real-time system that is used in most electronic railway interlocking systems from SIEMENS. For the verification bounded model checking algorithms are applied, i.e. a set of properties is formally proven. The completeness of this set is also determined efficiently.

1. Introduction

Today's electronic systems become more and more complex. They are applied in many areas of our personal lives, e.g. cellular phones or entertainment electronics, and it is nearly impossible to imagine a modern life without them. Failures of these devices would normally result just in minor problems for the users. But complex electronic systems are also used in safety-critical areas like medical equipment, avionics and electronic railway interlocking systems. Especially in these sectors, the demand for safety is exceptionally high, because human life may depend on the error-free functionality of such devices.

Nowadays railway systems are designed and tested in a very conventional way, i.e. the systems are simulated with a manually created test bench. The advantage is that the designers have a considerable expertise with this kind of work, but there is still a lot of potential for human failure. Furthermore, testing is very cost-intensive and can never reach complete coverage for large or complex designs. Thus, components like *Counting Heads* (CHs) [8], which are used in railway specific applications, have to be formally verified. These real-time systems are applied in safety-critical systems. They are used to determine whether a specified *Track Vacancy Detection Section* (TVDS) is clear or occupied. Especially for electronic railway interlocking systems, e.g. as constructed by SIEMENS, the correct function of CHs is crucial. An electronic interlocking system determines automatically, whether a TVDS is clear or occupied. But if the CHs fail to work properly a TVDS would either be falsely indicated as occupied – resulting in a deterioration of availability and reliability – or falsely indicated as clear – possibly introducing a safety hazard.

The interlocking system has to be verified to avoid such dangerous situations. In order to do this, our approach is to begin with the verification of the basic buildings blocks of such systems, namely CHs. Hence, we present the formal verification of CHs in this paper. The verification is done in two steps:

1. Properties are written for each signal, which is necessary in the verification process. These properties are proven to hold for the system using *Bounded Model Checking* (BMC).
2. The set of properties for each signal is proven to be complete, i.e. the properties cover all possible scenarios for this signal. The proof is done using the methods presented in [6].

In this way the correctness of the system can be completely verified. A CH works in two steps. The first one is counting axles, the second one is the evaluation of the counters. The verification of the system up to its evaluation phase is presented in this paper.

This paper is structured as follows: In Section 2 a brief introduction into the *Property Specification Language* (PSL) is given. Furthermore, the fundamental ideas of BMC and coverage proofs of properties are introduced in this section. We motivate the usage of CHs in railway systems in Section 3. Afterwards, the formal verification of CHs is discussed in detail in Section 4. The verification section is divided into two sections. In Section 4.1 properties are presented and explained. The proof that these properties cover all possible scenarios is given in Section 4.2. The final Section 5 concludes this paper and gives an outlook to future work.

2. Preliminaries

In this section we give some details regarding the preliminaries of formal verification. Although, the underlying system is modeled using SystemC, we omit details about SystemC and refer the reader to [5, 10].

2.1. Property Specification Language

Properties of a system are often described using temporal expressions in hardware design. With these properties

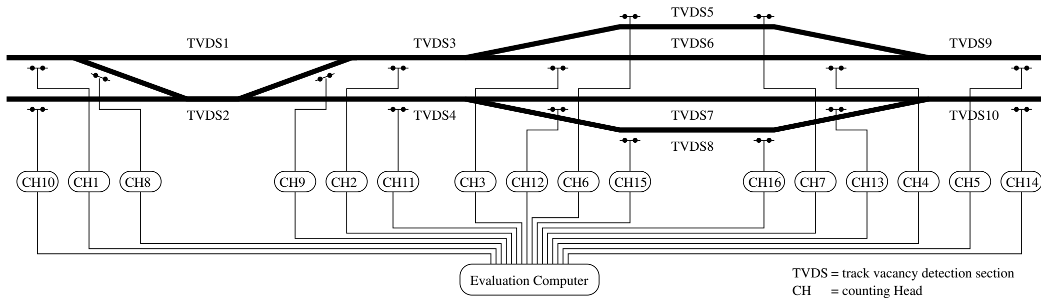


Figure 1. Track Vacancy Detection Section [9]

the system can be verified using formal techniques. Describing temporal properties for verification can be done in many different ways, since there exist several languages and temporal logics. As property specification language we use a subset of the widely known industrial standard PSL [1] from Accellera. A PSL-property, as used here, consists of two parts: a list of assumptions and a list of commitments. Assumptions and commitments have the form:

```
    next[a]       (expression)
or  next_a[a..b]  (expression)
or  next_e[a..b]  (expression)
```

where a and b are time points. If all assumptions hold, all commitments in the proof part have to hold as well. All used time points have to be greater or equal to 0. Since a and b are finite, a property argues only over a finite interval, which is called *observation window*. Temporal dependencies are expressed by using the keywords `next`, `next_a` and `next_e`, whereas `next_a` states that the expression has to hold at all time points in the interval and with `next_e` the expression has to hold at least once in the specified interval. Also a set of advanced operators and constructs is provided to allow for expressing complex constraints more easily.

In the way we formulate properties, they state that whenever some signals have a given value, some other (or the same) signals assume specified values. Of course, it is also possible to describe symbolic relations of signals. Furthermore the property language allows to argue over time intervals, e.g. that a signal has to be stable in a specified interval.

2.2. Bounded Model Checking

In *model checking* (also called *property checking*) properties for a given system are formulated in a "dedicated verification language". In this paper PSL is used as verification language. It is then formally proven whether these properties hold under all input and state assignments for the given assumptions. While "classical" CTL-based model checking [3] can only be applied to medium sized designs, approaches based on *Bounded Model Checking* (BMC) as discussed in [2, 11] give very good results when used for complete blocks. In BMC the properties are only considered over a finite interval. BMC has originally been proposed for circuit verification and in this context considering a finite number of steps is reasonable.

In order to verify a property the model has to be unrolled `n` times, where `n` is the maximum time interval of the property (the observation window). The unrolled model and the property are translated into a Boolean formula. The formula is solved by a SAT-solver and if it is satisfiable a counter example has been found, disproving the validity of the property. If the SAT instance is unsatisfiable the property holds. Since there is no restriction to reachable states during the proof of the corresponding SAT instance a counter-example may start from an unreachable state. Usually, if such a case occurs these states are excluded by additional assumptions.

2.3. Functional Coverage

So far the completeness of a property set is only ensured in a manual review by a verification engineer. Recently an approach to estimate functional coverage for BMC has been proposed in [6]. First, there have to be several properties describing the behavior of a particular signal or output. Second, after proving the correctness of these properties using BMC, the completeness of the set of properties is shown. The proof either verifies the coverage of the properties or provides a counter-example. Thus, an uncovered scenario, which has to be analyzed, can be derived.

The basic idea is to generate a new property, a so called *coverage property*, for each considered signal or output. If this coverage property is valid there is no scenario, in which the value of a signal or an output is not specified by the properties. It is proven that the union of all properties regarding a particular signal or output do not admit a behavior different to the one defined by the model. This is done by introducing a multiplexor for each bit which is driven by the signal or output and the inverted value of it. Afterwards, the coverage check is performed by proving that the multiplexor is forced to select the original value of the signal or output, assuming all involved properties. Examples are shown in Section 4.2. For a detailed description see [6].

3. Counting Heads

This section gives a short summary about the functionality and the modelling of *Counting Heads* (CHs) to keep this paper self-contained. For a more detailed description on the SystemC model see [7].

CHs [8] are needed to determine whether a railway track section is vacant or occupied. This is essential for electronic railway interlocking systems in order to position points into the correct direction for the next train. CHs are used in a lot of interlocking systems all over the world. A CH failing to work properly could result in a collision of railways and endanger the life of passengers.

In Figure 1 a small network of railtracks is shown, which could be operated by an electronic railway interlocking system. Every *Track Vacancy Detection Section* (TVDS) is de-

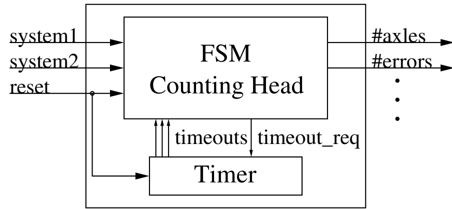


Figure 2. Abstracted Model of a CH

fined by at least two CHs, one at the beginning and one at the end of such a track section. The output of the CH is transmitted to an evaluation computer. The evaluation computer interprets the signals, compares the number of axles that entered and left a TVDS, issues clear or occupied indications and monitors the clear/occupied state of the TVDS.

3.1. Model

The CH is implemented using SystemC, a C++ class library. To model this system it is advisable to partition the design into a module for the *Finite State Machine* (FSM) and a module for the timeout control. The partitioning is necessary because a timeout can be triggered in any state. An abstracted model of the CH is shown in Figure 2. In the following we give some details about the model of the CHs that is verified later on.

As can be seen in Figure 2 the model has three input signals. The inputs `system1` and `system2` indicate which sensor system of a double wheel detector is affected and which is not. How a double wheel detector works and its influence on the system is explained in Section 3.2.1. The input `reset` is used to reset the system into its initial state. The inputs are Boolean values.

As mentioned above the system consists of two modules. These are interconnected to each other by three timeout signals and one timeout request signal. The timeout request to start the timer is issued every time the module "FSM-Counting Head" reaches another state in the FSM (see Figure 4 and Section 3.2.2). The timer module activates the timeout signals, when the corresponding timeout is reached. The internal timing signals are Boolean values, too.

The number of outputs depends on the type of the CH. But all CHs have at least the following outputs:

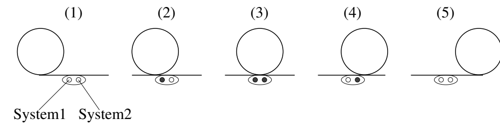
- Number of axles, integer value (The sign indicates the direction of the axles.)
- Number of errors, unsigned integer value
- Time the system has been unaffected (idle time), unsigned integer value
- Counter control token, Boolean value

Additionally, there can be several more outputs, called failure tokens, depending on the specific needs for a railroad track.

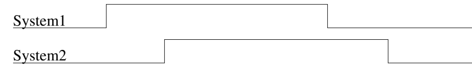
If a timeout is triggered in any state, the system starts its evaluation phase. In this phase the system determines how many axles crossed the double wheel detector, how many errors occurred and if the control token has to be set. During this phase all outputs are written.

3.2. Axle Counting Procedure

In this section the basics for the axle counting procedure are presented. This is necessary to understand the formal verification in Section 4. First, the double wheel detector and its influence on the FSM is clarified in Section 3.2.1. Second, the FSM itself is discussed in Section 3.2.2.



(a) Impact on Sensor Systems



(b) Waveform

Figure 3. Double Wheel Detection [8]

3.2.1. Double Wheel Detectors

A double wheel detector is mounted on one of the rails. It detects passing wheels of the vehicles. A double wheel detector consists of two sensor systems, which are triggered when an axle crosses. The impacts on both sensor systems occur with a delay, which indicates the direction of the crossing axle. A regular crossing of an axle from left to right is illustrated in Example 1.

Example 1. *The triggering of the sensor system is shown in Figure 3(a). The signal waveform of both sensor systems is given in Figure 3(b). These figures correspond to an axle which crosses the double wheel detector regularly.*

If the space between wheel and sensor system is big enough, both sensor systems are unaffected (state (1)) as can be seen in Figure 3(a). The wheel approaches from the left side and the left sensor is affected first (state (2)). When the wheel is positioned right above the double wheel detector, both sensor systems are affected (state (3)). The axle moves on and the left sensor is not affected anymore (state (4)). Finally, the axle left the double wheel detector and both systems return to their unaffected states (state (5)).

The output of the double wheel detector is taken as an input for the model shown in Figure 2. These two inputs determine the state transition to the corresponding state in the FSM.

3.2.2. Finite State Machine

The basic idea for the axle counting procedure is as follows: There are five counters (`l`, `r`, `g`, `x`, `a`). They count a defined type of state transition for a single axle, while traversing the FSM given in Figure 4. All counters are integer valued.

Every time the FSM enters state `unaffected`, i.e. both sensor systems of the double wheel detector are unaffected, these counters are added to a second set of counters, which are counting the whole group of axles. Afterwards, the counters for a single axle are set to zero.

The FSM consists of four states. They correspond to the four possible impact combinations on both sensor systems. The states, in which one sensor system is exclusively affected, are divided into two substates `was_both_systems` and `was_unaffected`. Depending on whether the CH was in state `both_sys-`

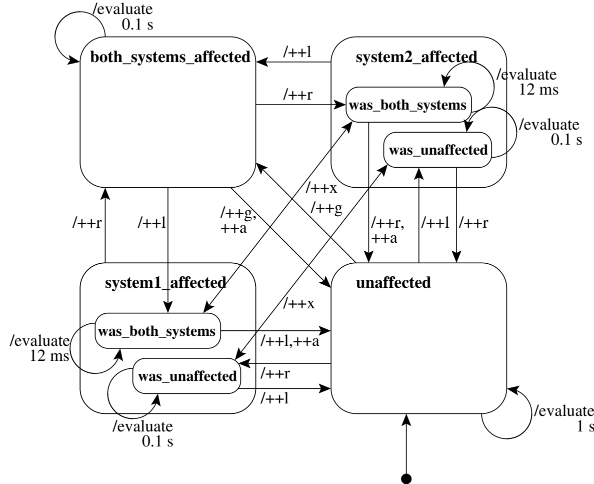


Figure 4. FSM of the Counting Head [8]

tems_affected or in state unaffected before entering this state.

The states of the FSM are traversed in a circle either clockwise or counterclockwise, depending on the direction in which the axle crosses the detector. For a better understanding, two possible state transitions are illustrated. A transition which could occur in a regular passing of an axle is shown in Example 2 and a transition which indicates a irregular passing of an axle is presented in Example 3.

Example 2. *Assume the system is in state* system1-_affected *and in substate* was_both_systems*, i.e.* system1 *of the double wheel detector is affected and* system2 *is not. If the impact on* system1 *stops, a state transition to state* unaffected *occurs. During this transition the counters* a *and* r *are incremented.*

Example 3. *Assume none of the sensor systems is affected and in the next time step both sensor systems are affected. This means that a state transition from state* unaffected *to state* both_systems_affected *occurs. During this transition only the counter* g *would be incremented.*

The remaining transitions can be seen in Figure 4. The counters x and g are necessary to determine whether an axle crossed the detector regularly. Without interferences the counters x and g stay equal to zero.

At some of the states in Figure 4 there is a time annotated. The time intervals are ranging from 12 ms to 1 s and they are not given in form of discrete clock cycles as usually known in hardware design. These intervals indicate the time the CH has to stay in that particular state, before a timeout is triggered and the counter values are evaluated. In the evaluation phase the counting head determines how many axles have passed the detector. The direction of these axles is calculated and whether irregularities occurred is determined. The results of the evaluation phase are written to the corresponding outputs. The verification of the evaluation phase is not discussed in this paper. Hence, we refer to [8] for a detailed description.

All calculations and increments of counter values have to be done in real-time. This is important since the railway interlocking system has to be up-to-date at any time.

4. Formal Verification

A reliable implementation is essential for safety-critical systems, as electronic railway interlocking and its basic building blocks. For these systems, which also operate time-critical, it is very complex to prove the correct behavior in any circumstance. For example, for a CH the time intervals, as annotated in Figure 4, have to be considered. In this paper, the formal verification is done by an inductive proof strategy combined with BMC. Without BMC the proofs would most probably fail, because of the state explosion problem known from conventional model checking algorithms.

In Section 4.1 we present properties for the formal verification of all internal signals which are necessary to prove the correctness of the system from its initialization up to the beginning of its evaluation phase, i.e. state variables, counter values and signals for timeouts and timeout requests. Afterwards, in Section 4.2 we show the proof of completeness of these properties. In this way, we prove the correctness of the model from the time the system is initialized until the evaluation phase begins. Finally, the correctness of the evaluation phase has to be proven, but this part of the formal verification is not discussed here.

4.1. Verification by BMC

In this section we explain details about the formal verification of properties on state variables, counter values and timing issues. Most of the proofs are done according to the inductive principle:

1. *Prove* the correctness of the system after initialization.
2. *Assume* this correctness at time point t.
3. *Prove* that the system is still correct at the next time point $t + 1$.

Now, it is very important that all counter values have got the expected values at the beginning of the evaluation phase, since they are used to evaluate the system. The counter values are determined by traversing the FSM, as already described in Section 3.2.2. Thus, we have to prove that the FSM is implemented correctly according to its specification and we have to prove that the system actually reaches the evaluation phase. Both parts are shown in the following sections. To prove the the properties as valid or invalid the SystemC property checker *CheckSyC* [4] is used.

4.1.1. Verification of the FSM

Basically, two steps have to be considered to verify the correctness of the FSM. These steps are explained in the following paragraphs.

Relations between states and inputs The first step is to relate the states of the FSM to the inputs system1 and system2. This is done by five properties. There are three of these properties given in Figure 5. As can be seen the properties start with a triggered reset (lines 1 – 6). This property states that if the reset is set to 1 (line 3) the state is set to unaffected at the next time point (line 5). Note, a state set to 0 means the state is unaffected, 1 means system1_affected, 2 means system2_affected and 3 means both_systems_affected. The second property (lines 8 – 14) states that, if both sensor systems

```
1   property trans_1_0 =
2     always (
3       state == 1 && system1 == 0 && system2 == 0 && reset == 0
4       && r < 65535 && l < 65535 && x < 65535 && g < 65535 && a == 0
5     ) -> (
6       next[1](state == 0
7         && g == prev[1](g) && x == prev[1](x) && l == prev[1](l) + 1 && r == prev[1](r)
8         && state_old == prev[1](state_old) && ((state_old == 0) ? (a == prev[1](a))
9                                                                  : (a == prev[1](a) + 1)))
10    );
```

Figure 7. Property for a State Transition

```
1   property state_reset =
2     always (
3       reset == 1
4     ) -> (
5       next[1](state == 0)
6     );
7
8   property state_0 =
9     always (
10      system1 == 0 && system2 == 0
11      && reset == 0
12    ) -> (
13      next[1](state == 0)
14    );
15
16  property state_1 =
17    always (
18      system1 == 1 && system2 == 0
19      && reset == 0
20    ) -> (
21      next[1](state == 1)
22  );
```

Figure 5. Properties for `state`

```
1   property state_0_change =
2     always (
3       system1 == next[1](system1)
4       && system2 == next[1](system2)
5       && state == next[1](state)
6       && next_a[0..1] (reset == 0)
7     ) -> (
8       next[1](state_old)
9         == next[2](state_old)
10    );
```

Figure 6. Property for `state_old`

are unaffected (line 10) and the `reset` is 0, the system will be in state `unaffected` in the next clock cycle (line 13). There are also properties for the remaining combinations of the inputs `system1` and `system2`. These are formulated in a similar way. Thus, there exist six properties in total. At this point we assume these properties to cover the behavior of the variable `state` completely. The proof is shown in Section 4.2. Now we can use the variable `state` in the assume parts of following properties.

The correct value of `state_old` has to be verified, too. This variable determines the substates in the states `system1_affected` and `system2_affected`. Exemplarily, we give one property for this variable in Figure 6. This property states that if nothing happens, i.e. `system1`, `system2` and the `state` do not change (lines 3 – 5), `state_old` does not change either (lines 8 – 9). The property may seem trivial, but in most cases this is one of the properties which is forgotten. If this property is missing in the coverage proof, a counter-example would be provided with exactly the scenario where the behavior is not specified. Of course, there are more properties for this signal defining its remaining behavior.

The proofs for the completeness of the properties presented here will be given in Section 4.2.

Relations between state transitions and counters In this section the values of the counters are related to the state transitions, i.e. it is proven that the counters are incremented according to the annotation given in Figure 4. There are 17 properties to cover all possible state transitions:

1. Three properties for every state transition from each state.
2. For each state one property for the case that no changes occur.
3. One property for the reset condition.

In Figure 7[1] there is the property given which formulates a state transition from state 1 (`system1_affected`) to state 0 (`unaffected`) (line 3), similar to Example 2. The assumption `state == 1` states that the FSM is in state `system1_affected` and the transition to state `unaffected` is enforced by `system1 == 0` and `system2 == 0`. The remaining assumptions (line 4) are given to avoid false negatives and overflows of the counter values. To constrain the counters like that is valid, because they are considered to have a low value. According to the specification CHs are only used in environments, where the values of these counters are in a reasonable interval. In the list of commitments (proof part, lines 6 – 9) the first statement is that state `unaffected` (0) (line 6) is reached. According to Figure 4 all counters keep their values except for the counter `l` and the counter `a` (lines 7 – 8). An increment of the latter counter depends on the previous state. If the previous state was `unaffected` (substate `was_unaffected` in Figure 4) no axle was counted and the counter `a` keeps its value (end of line 8). Otherwise, the system has been in state `both_systems_affected` and the axle counter `a` is incremented by 1 (line 9).

The properties for the remaining state transitions are formulated in a similar way according to the FSM-specification. The property for the reset condition is straightforward, i.e. after the `reset` is triggered every variable is set to 0. The remaining four properties state that if there is no alternation on the inputs `system1` and

[1]The conditional operator `?`, known from C, is used here with same meaning as it has in C, i.e. `(X)? Y : Z` states `if (X) then Y else Z`.

```
1   property timer_eval =
2     always (
3       evaluation == 0
4       && reset == 0
5       && (state == 1 || state == 2)
6       && state_old == 1
7       && ttrail_signal == 1
8     ) -> (
9       next[1](evaluation == 1)
10  );
```

Figure 8. Property for `evaluation`

`system2`, there is no alternation in any of the counter or state variables.

Finally, each counter for a single axle has to be related to its corresponding counter for a group of axles. This is necessary since the group counters are used in the evaluation phase. But the details for this proof are left out due to page limitation. The inductive proof needs three properties for each counter:

1. One property for the reset state.
2. One property for the induction step.
3. One property for the case that the system state remains the same.

4.1.2. Timeouts and Evaluation

If a timeout is triggered, the system has been idle in a particular state for a specified time interval. If this happens the evaluation phase begins, as already said in Section 3. The timeout intervals range from 12 ms up to 1 s.

To verify the correct behavior of the system until the beginning of the evaluation phase, the timer module from Figure 2 has to be verified. This is done with five properties and an inductive proof strategy. In the top module the variable `evaluation` has to be related to the timeouts coming from the timer module. Exemplarily, this is done for two states with the property shown in Figure 8. This property states that the system starts the evaluation phase in the next clock cycle (line 9), if

1. the system is not already in the evaluation phase (line 3),
2. the system is not in the initial state (line 4),
3. the system is in state `system1_affected` (1) or `system2_affected`(2) (line 5),
4. the previous state (`state_old`) was `both_systems_affected` (line 6),
5. the timeout `ttrail` ($12ms$) was triggered.

There is one property for each state and substate in which a timeout can be triggered. Thus, there are four properties to prove `evaluation`. Of course, there is a last property which formulates that if no timeout is triggered, the evaluation phase does not begin. Finally, there are 10 properties to prove the correct beginning of the evaluation phase (including the necessary timeouts).

4.2. Coverage Analysis and Proving Completeness

In this section the completeness of the property set is shown. Several properties have been formulated and presented in the previous section. Internal signals, e.g. `state`,

```
1   property coverage_state =
2     // @insertMuxForSignal: state select
3     always (
4       //reset
5       ((reset == 1) ?
6                   next[1](state == 0)
7                 : 1)
8       &&
9       //state_0
10      ((system1 == 0 && system2 == 0
11        && reset == 0) ?
12                  next[1](state == 0)
13                : 1)
14      &&
15      //state_1
16      ((system1 == 1 && system2 == 0
17        && reset == 0) ?
18                  next[1](state == 1)
19                : 1)
20      && ...
21      // coverage transformations for the
22      // two remaining states properties
23      && (select == 1)
24    )->(
25      next[1](select == 1)
26    );
```

Figure 9. Coverage Property for `state`

were proven by a set of properties. In order to use the signals in assumptions of other properties, the property set has to be proven to cover every possible scenario. During the proof of completeness using the approach [6], counter-examples can occur. Due to the nature of this approach these counter-examples describe uncovered scenarios.

4.2.1. Coverage Property

To prove the completeness, the given properties for one particular signal have to be transformed into a coverage property. For example, the properties from Figure 5 are transformed to a new property, shown in Figure 9. To do this, all properties which argue over the considered signal have to be identified and the maximum time point has to be determined. Afterwards, each property is syntactically transformed from the expression in Figure 5 (e.g. lines 2 – 6) to an expression which can be assumed (lines 5 – 7 in Figure 9). Note, the transformation is straightforward and does not change the semantic of the property. In this way all properties are reformulated and assumed in the coverage property (lines 9 – 22). To complete the coverage property a multiplexor has to be inserted into the model. This is done by the special command in line 2. The multiplexor is inserted at the considered signal and the input of the multiplexor is named `select`. Finally, `select` has to be assumed to be true at all time points except the maximum time point (line 23). The proof part consists of the commitment `select == 1` at the maximum time point `1` (line 25). If this property holds, the signal is completely covered by the properties presented in Section 4.1.1, i.e. there is no uncovered scenario. The coverage property in Figure 9 holds. That a coverage property identifies uncovered scenarios is shown in detail in [6]. Such a coverage property has to be generated for each signal needed to verify the correctness of the CH model, starting from the initial state up to the beginning of the evaluation phase.

Table 1. Number of Properties

#	Signal	#p	covered?	#p added
1	state	6	yes	-
2	state_old	8	yes	-
3	state transitions	17	yes†	-
4	for each counter	3	yes	-
5	timeouts & evaluation	10	yes‡	5

† excluded states ‡ added properties

```
1  next_a [0..2](
2        x < 65535 && g < 65535
3    && l < 65535 && r < 65535)
```

Figure 10. Added Assumptions

A summary of the signals and the number of needed properties is given in Table 1. The name of the verified signal is given in column *Signal* and a related ID is shown in column *#*. The number of required properties is listed in column *#p*. The result of the coverage proofs is presented in column *covered?*. Note, the coverage properties for the signals in the rows marked with † or ‡ could not be proven as valid in the first attempt. For both signals uncovered scenarios in form of counter-examples have been found. These coverage gaps are discussed in the next section and possible solutions to close them are presented.

4.2.2. Coverage Analysis

If a counter-example occurs it has to be analyzed. Finding a solution is well supported, since a counter-example directly corresponds to an uncovered scenario. There are two possible solutions to close this verification gap:

1. Adding properties
2. Excluding states

There are two different situations where one of these two solutions are applied. The first solution is always used, if the analysis of the counter-example indicates that the set of properties was incomplete, i.e. a real scenario has been discovered in which the behavior of the considered signal was not specified. The second solution is applied, if the specification did not specify the behavior of a signal in a special situation, e.g. this situation is excluded by reason from the very beginning.

State transitions For the coverage proof of the state transitions (line 3 of Table 1) states had to be excluded. After trying to prove the coverage property an counter-example was provided. The source for this counter-example was an overflow situation of counters of single axles, which was not covered in the properties.

As can be seen in the property in Figure 7 (line 4), the counters for single axles are assumed to be less than 65535 to prevent an overflow. This is valid because the counters for single axles are assumed to have low values, e.g. a regular passing of an axle would result in either `r = 4` or `l = 4`. Hence, there is no overflow handling in the underlying model. But this is exactly the scenario which the coverage approach detects as uncovered. As already said in Section 4.1.1 the CHs are used in an environment which ensures that the counters have got reasonable (low) values. Hence, these scenarios can be excluded, by adding the assumptions shown in Figure 10 to the coverage property.

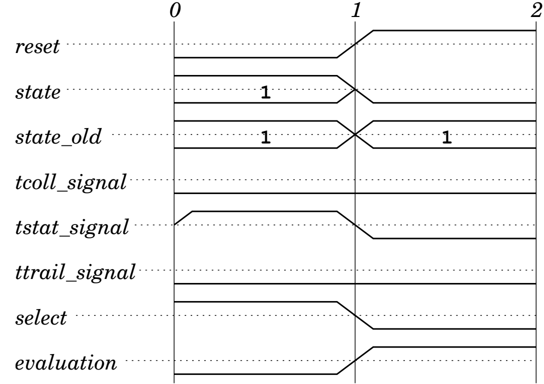


Figure 11. Trace for a Counter-Example for `evaluation`

Timeouts and evaluation Since a triggered timeout results in the beginning of the evaluation phase, the timeouts and `evaluation` are proven and analyzed together. The properties for the timeout signals covered all scenarios, thus they are not considered in the following.

Amongst others, the proof of coverage for `evaluation` provided the counter-example shown in Figure 11. The scenario presented here shows a situation which was not covered by any property before. In the shown trace it can be seen, that the system is in state `system1_affected` (`state = 1`) and was in state `both_systems_affected` (`state_old = 1`). The only timeout, according to the FSM in Figure 4, which would result in the beginning of the evaluation phase is the 12 ms timeout (`ttrail_signal`). This signal is 0 as can be seen in this trace. But another timeout `tstat_signal` is indicated as triggered (100 ms). This timeout, also the 1 s timeout (`tcoll_signal`), has no influence on the behavior of the system in this particular substate. So far there was no property to cover this state. Thus, the property shown in Figure 12 had to be added to the set of properties and to the coverage property. In comparison to the property in Figure 8 this property proves that the evaluation phase does not start (line 11), if the system was in the following state:

1. `state` was in state `system1_affected` or `system2_affected` (line 5).
2. `state_old` was in state `was_both_systems` (line 6).
3. The timeout `ttrail_signal` (12 ms) was not triggered (line 9).
4. Any of the two remaining timeouts is triggered (lines 7 – 8).

The property from Figure 12 closes the gap provided by the coverage proof. But there are still situations which are not not described by a property. These are:

1. The system is in state `system1_affected` or `system2_affected` **and** the system was in state `unaffected` **and** the 100 ms timeout was not triggered, but one of the two remaining timeouts is triggered.

```
1   property timer_eval_1 =
2     always (
3       evaluation == 0
4       && reset == 0
5       && (state == 1 || state == 2)
6       && state_old == 1
7       && (tcoll_signal == 1
8           || tstat_signal ==1)
9       && ttrail_signal == 0
10    ) -> (
11      next[1](evaluation == 0)
12  );
```

Figure 12. Added property to cover `evaluation`

2. The system is in state `both_systems_affected` **and** a timeout, except the 100 ms timeout, is triggered.
3. The system is in state `unaffected` **and** a timeout, except the 1 s timeout, is triggered.
4. The evaluation phase begins exclusively if a timeout was triggered. Otherwise, the evaluation phase does not begin.

For each of these situations one property similar to the property in Figure 12 was added to the set of properties. Therefore, five properties, as indicated in Table 1, were added to verify `evaluation` and proving the completeness of the verification.

4.3. Run-Time and Memory Consumption

In Table 2 a summary of run-times and memory consumptions is given. All properties were proven on a computer with an AMD64 3500+ CPU and 1 GB of main memory running under Linux.

In the first two columns an ID, column *#*, and the name of the signals, column *Signal*, is given. Column *BMC* is divided into three columns. In column *Time* the overall CPU-time used to prove all properties for the considered signal is given. The number of properties is shown once again in column *#p*. The maximum memory consumption is presented in column *Memory*. Maximum memory consumption means that no single proof needed more memory than the given number in this column. The 61 BMC properties have been proven to hold in a total of 432.22 s. *CheckSyC* did only need 257 MB of main memory to verify the properties.

The column *Coverage* is divided in the same way as column *BMC*. For each signal one coverage property is needed. But for the counters five coverage properties, one for each single counter, have been necessary, as can be seen in column *Coverage - #p*. Thus, the total of coverage properties is nine. Each coverage property was proven in approximately 9 s on average. All coverage properties have been verified using less than 300 MB of main memory.

In total, the time to prove all BMC- and all coverage properties was 510.8 s and the maximum memory consumption was lower than 300 MB. The formal verification of CHs has been accomplished in a moderate effort with regard to hardware resources and run-time.

Table 2. Run-Time and Memory Consumption

		BMC			Coverage		
#	Signal	#p	Time	Memory	#p	Time	Memory
1	state	6	43.40 s	254 MB	1	7.80 s	254 MB
2	state_old	8	57.97 s	254 MB	1	7.83 s	255 MB
3	state transitions	17	122.69 s	257 MB	1	8.69 s	257 MB
4	counters	15	103.74 s	216 MB	5	45.16 s	294 MB
5	timeouts & evaluation	15	104.42 s	254 MB	1	9.10 s	293 MB
	overall	61	432.22 s	257 MB	9	78.58 s	294 MB

5. Conclusions and Future Work

We have presented the formal verification of the axle counting system up to the evaluation phase of Counting Heads for railways. Therefore, the correctness of the system has been proven using BMC and an inductive proof strategy. The proofs range from the system initialization to the beginning of the evaluation phase. Afterwards, the completeness of the presented set of properties was analyzed. Based on an automatic approach the completeness of the set of properties was shown. After closing coverage gaps the number of properties is 61 in total. These proofs have been managed in a few minutes and a moderate usage of hardware resources.

Thus, we verified the correct behavior of CHs, a real-time system which is used in a safety-critical environment, until the beginning of the evaluation phase. Based on the results of this paper, in future work we will formally verify the evaluation phase. Thereby, the already verified counters can be assumed in the necessary proofs. Thus, the verification effort for the evaluation phase is reduced.

References

[1] Accellera. *Property Specification Language Version 1.1*, 2004.

[2] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conf.*, pages 317–320, 1999.

[3] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential circuit verification using symbolic model checking. In *Design Automation Conf.*, pages 46–51, 1990.

[4] R. Drechsler and D. Große. System level validation using formal techniques. *IEE Proceedings Computer & Digital Techniques, Special Issue on Embedded Microelectronic Systems: Status and Trends*, 152(3):393–406, May 2005.

[5] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[6] D. Große, U. Kühne, and R. Drechsler. Estimating functional coverage in bounded model checking. *Design, Automation and Test in Europe*, 2007.

[7] S. Kinder and R. Drechsler. Modeling and formal verification of counting heads for railways. *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)), Braunschweig, Germany*, 2007.

[8] Siemens AG. Az S M Multiple-section Axle Counting System. Copyright, Siemens AG, 2003.

[9] Siemens AG, Transportation Systems, Rail Automation. Safety for the Rail Services. www.siemenstransportation.co.uk/pdfs/AzSM%20R.pdf.

[10] Synopsys Inc., CoWare Inc., and Frontier Design Inc., http://www.systemc.org. *Functional Specification for SystemC 2.0*.

[11] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. Cost-efficient block verification for a UMTS up-link chip-rate coprocessor. In *Design, Automation and Test in Europe*, volume 1, pages 162–167, 2004.