Uniform Modeling of Railway Operations

Eduard Kamburjan and Reiner Hähnle

Department of Computer Science, TU Darmstadt, Germany
{kamburjan,haehnle}@cs.tu-darmstadt.de

Abstract. We present a comprehensive model of railway operations written in the abstract behavioral specification (ABS) language. The model is based on specifications taken from the rulebooks of Deutsche Bahn AG. It is statically analyzable and executable, hence allows to use static and dynamic analysis within one and the same formalism. We are able to combine aspects of micro- and macroscopic modeling and provide a way to inspect changes in the rulebooks. We illustrate the static analysis capability by a safety analysis based on invariant reasoning that only relies on assumptions about the underlying railway infrastructure instead of explicitly exploring the state space. A concrete infrastructure layout and train schedule can be used as input to the model to examine dynamic properties such as delays. We illustrate the capability for dynamic analysis by demonstrating the effect that different ways of dealing with faulty signals have on delays.

1 Introduction

Railway systems are a domain where formal modeling of systems and formal analysis methods are generally accepted by industry and partially required by certification authorities [3]. Therefore, the railway domain is an active and important area of applied research in formal methods.

Models of railways can be classified according to their level of abstraction and their intended degree of analyzability. Regarding the *abstraction level*, modeling approaches tend to be either *microscopic* or *macroscopic*. The former focus on modeling a local part of a railway network, e.g., a few train station to be as precise enough to examine local and detailed properties. On the other hand, macroscopic models aim to be sufficiently abstract to cover a large part of the whole network to analyze global or coarse properties. Regarding *analyzability*, current models concentrate on a single aspect only, e.g., the safety of interlocking and signaling systems or the network throughput.

Railways are complex systems whose global properties such as safety or capacity are determined by low-level structural components as well as by communication protocols between stations at a high abstraction level. Failures of the infrastructure happen at the component (i.e., low) level, but they have global impact, e.g., a faulty signal introduces delays that are not analyzable in a model that abstracts away from individual signals.

To reconcile different levels of abstraction, we propose a uniform modeling approach that is flexible enough to capture and analyze a wide range of properties. This uniformity has important advantages:

1. The overall effort of modeling is reduced, because each aspect needs to be modeled only once.
2. Aspects from macro- and microscopic modeling can be represented in a single model.
3. Hence, it is possible to analyze the effects that perturbations at a low abstraction level have on the global, system-wide behavior.

Our modeling method is based on the ABS language [18], which was originally designed to model and analyze concurrent/distributed *software* systems. We argue that its concurrency and object model are a good match for railway systems, too. We substantiate our claim by performing two complementary kinds of analysis carried out with one and the same model:

Dynamic analysis of runtime behavior. ABS models are executable. We demonstrate how a change in the rules for handling faulty signals influences the travel time of a train passing this signal. To do so, we simulate the scenario and compare the generated event traces. The example is based on a fault in a single signal, but the rules to handle this case involve up to three different train stations and two trains. The fault is only observable at a microscopic modeling level, but its effects have a global impact.

Static analysis of a global safety property. We prove that on a single line between two stations it is never the case that there are two trains announced in opposing direction. Our analysis is based on *deductive* invariant reasoning and not on model checking. We analyze the *communication structure* between trains, infrastructure and station, so we are able to state safety *independently* of a concrete track plan, as long as that is well-formed.

We do not verify implementation details of the structural components such as correctness of interlocking tables, but assume other, well-established methods have checked these. We concentrate on procedures and communication, and how a fault is handled *on the operational level*. E.g., we do not model the internal behavior of the signal once it broke, but we model precisely, how the mitigating communication between stations and trains in the signal's proximity ensure safety. Such procedures are described in detail in the *Fahrdienstvorschrift* [6] for all railways in Germany operated by Deutsche Bahn AG. Our model is a partial formalization of the description of ETCS 1LS within that rulebook. Our main contributions are:

1. A novel, uniform modeling approach of railways in the concurrent, executable language ABS that allows static and dynamic analysis.
2. A deductive invariant-based analysis of safety of railway communication.

The paper is organized as follows: In Section 2 we present the ABS language and in Section 3 our model of railway operations. In Section 4 we show how changes in procedures can be analyzed by simulation. In Section 5 we show a safety property and show how ABS admits its formal proof. Related work is in Section 6 and we conclude in Section 7.

2 ABS

ABS is an object-oriented, executable modeling language designed to model concurrent and distributed software systems [18]. Its syntax is loosely based on Java and most concepts of ABS are (intentionally, to ease its usage) standard. We refrain from introducing the whole language, instead we focus on three of its distinguishing features that are relevant in the present context: The concurrency model based on asynchronous method calls, explicit modeling of time, and formal semantics. A full introduction can be found in [11, 18].

ABS models can be compiled into executable Erlang, Java, Maude, ProActive or Haskell code. In this case an *initialization block* must be provided (not necessary for deductive static analysis). This is a special ABS statement that serves as the entry point of a model. While ABS classes describe general behavior, the initialization block sets up a scenario.

2.1 Concurrency Model

ABS extends the actor [14] paradigm: Objects on different processors do not share memory. Each processor may host several objects from different classes. Even though ABS permits objects on the same processor to access shared memory, we carefully avoid this possibility in our model to render verification easier.

ABS objects are strictly encapsulated and have neither public nor static fields. Any inter-object communication is accomplished by asynchronous[1] method calls: The caller invokes a method and continues its own execution without waiting for the call to terminate. Instead, the caller has a *future* as a handle, which is used to wait for the called method (if necessary) and to read its return value.

Example 1. The following code calls method `m` on the object stored in `o` and saves the future in local variable `f` (line 1); it waits for `m` to terminate (line 3) and reads the return value into local variable `i` (line 4).

```
1 Fut<Int> f = o!m();
2 ... do something else ...
3 await f?;
4 Int i = f.get;
```

If there is no code between lines 1 and 3, then there is a shorthand notation for this idiom that avoids creation of an explicit future: `Int i = await a!m();`. □

[1] For abstraction of sequential computations there are synchronous calls as well.

Upon receiving the call, the callee object creates a new process and puts it into its process pool. For a process to become active, the currently active process on its processor must *explicitly* release control by termination or waiting. The statement `await g` releases control by the active process and waits for the guard `g` to become true. The guard `g` has one of the following forms:

- a future query `f?`, where the process can be reactivated after the process corresponding to this future has terminated;
- a side-effect free boolean expression (including future queries), where the process can be reactivated whenever the expression evaluates to true, e.g., `await this.counter > 5;`
- a time advancing expression as introduced in the next section.

The explicit release of control allows to reduce the number of interleavings between processes, since between the `await` statements, a process has exclusive control over the object memory and can be regarded as sequential.

The scheduler is non-deterministic, i.e., whenever more than one process can be reactivated, one of them is chosen non-deterministically.

2.2 Modeling Time

ABS allows to advance time explicitly [2] in processes. There are two statements to let time pass:

- `duration(t1,t2);` blocks the active process between `t1` and `t2` time units. ABS leaves open how long a time unit is — in this work we use seconds.
- `await duration(t1,t2);` suspends the active process between `t1` and `t2` time units. At runtime a number between `t1` and `t2` is randomly chosen. The process can be activated earliest after this time, but if other processes are active and consume time, it may take longer.

There is no global clock, each object has a local clock. The clock of an object is advanced if (1) it is the earliest local clock and (2) no process in any other object can advance its clock. The local time can be accessed with `now()`.

2.3 Four Event Semantics

The formal semantics of ABS can be described with the help of communication events, each describing a communication action of a process [8]. We use four different events, one for each possible action of a process that is visible to the outside: activation of the process, starting its execution, termination, and obtaining a value from a future. Whenever such an action occurs, the process appends the corresponding event to the global history. Note that when executing the model in a runtime environment, there is no such history, it is only used to define the semantics and reason about possible behaviors.

Definition 1 (Events). *Let O, O' range over object IDs, f over futures, e over expressions and m over method names. The symbol e^* denotes a possibly empty sequence of expressions and represents the parameters of a method call. Events* Ev *are defined by the following grammar:*

$$\begin{aligned}
\mathsf{Ev} ::= \ & \mathsf{invEv}(O, O', f, m, e^*) && \textit{(Invocation Event)} \\
| \ & \mathsf{invREv}(O, O', f, m, e^*) && \textit{(Invocation Reaction Event)} \\
| \ & \mathsf{futEv}(O', f, m, e) && \textit{(Resolution Event)} \\
| \ & \mathsf{futREv}(O, f, e) && \textit{(Resolution Reaction Event)}
\end{aligned}$$

An invocation event is added when O calls $O'.m(e^*)$ with future f as a handle. The invocation reaction event is added once O' starts the execution of this call. ABS assumes that the call is received at the same timepoint as the invocation, but not that it is immediately executed. The resolution event is added once the process which has f as its handle terminates with the return value e in object O'. The resolution reaction event is added once object O reads the value e from future f. Note that O is not necessarily the caller object, because f can be passed as an argument.

Every history h an ABS system produces is *well-formed*, satisfying certain conditions on the ordering of events. For example, if there is an $i \in \mathbb{N}$ with $h[i] = \mathsf{invREv}(O, O', f, m, e^*)$, then there must be a $j < i$ with $h[j] = \mathsf{invEv}(O, O', f, m, e^*)$. This condition expresses that every process starts its execution only after it was called. The well-formedness conditions for all event types are in [8].

Example 2. Assume that histories are axiomatized as a theory of finite sequences. Then we can express invariant properties over histories as formulas in first-order logic. For example, the property that for each object, between any two calls of method m there is a call of method m' can be written as the following formula:

$$\begin{aligned}
&\forall\, \mathsf{Object}\ O;\ \forall\, \mathsf{Int}\ i, j;\ i < j \rightarrow \\
&\quad \Big(\big(\exists\, \mathsf{Object}\ O', O'';\ \exists\, \mathsf{Fut}\ f, f';\ \exists\, \mathsf{Expr}^*\ e, e';\ \\
&\qquad \mathit{history}[i] \doteq \mathsf{invocEv}(O', O, f, m, e) \wedge \mathit{history}[j] \doteq \mathsf{invocEv}(O'', O, f', m, e') \big) \\
&\quad \rightarrow \big(\exists\, \mathsf{Int}\ k;\ i < k < j \wedge \exists\, \mathsf{Object}\ O';\ \exists\, \mathsf{Fut}\ f;\ \exists\, \mathsf{Expr}^*\ e; \\
&\qquad \mathit{history}[k] \doteq \mathsf{invocEv}(O', O, f', m', e)) \big) \qquad \square
\end{aligned}$$

Global history invariants can capture system properties and may reference the fields of any object in the system. An invariant must hold at each point when a process terminates or is suspended, hence it is sufficient to create proof obligations that are local to methods: Because of strong encapsulation, methods on one object have no direct access to the fields of other objects—to verify global invariants, these are split into local invariants that specify the object-local history. The KeY-ABS tool [7] is then able to statically and formally verify that each method in a class preserves its local history.

With invariant-based reasoning we are able to state properties of *all* histories realized by a system, while the execution of the ABS model generates only one

history. However, the four event semantics in [8] does not include the timed semantics of ABS and is thus not able to express properties concerning time. This is the subject of future work.

3 The Railway Operation Model

Our model is focused on operations and is derived from rulebooks. Not all components are described in the rulebooks, but also in requirement specifications or technical documents. For instance, the communication between stations is in part described in *Ril 408* [6] and in part by documents specifying the mechanisms for route blocks. We consider participating infrastructure elements as black boxes and only describe their behavior to extent that it is specified in the rules. If the rules do not fully specify component behavior, then we complete the behavior from the descriptions found in technical documents, but without implementation details. For example, we do not distinguish between mechanical and electronic interlocking systems.

We model physical behavior, including vehicle dynamics, with sufficient precision to establish capacity and safety properties. On the other hand, we simplify some scenarios which are either forbidden in the rulebooks or that have a negligible effect on the properties to be shown. For example, we compute braking distances using the track gradient, but we do not model how trains roll back a short distance after releasing their brakes.

Our model uses *instantaneous communication*—communication has no delay and is processed immediately, state changes take no time. In the future we plan to model such delays, but we expect this to be straightforward.

3.1 Infrastructure

We model the rail track plan as a graph, where nodes are fixed *points of information flow* and edges are tracks between these points.

Definition 2 (Point of Information Flow). *A* point of information flow *(PIF) is a position on a track where one of the following criteria applies:*

- *There is a structural element allowing a train to receive information, for example, a signal or a data transmission point of a train protection system.*
- *It has a critical distance in the direction of a signal: At this point the signal is seen at the latest (for example, according to* Ril 819.0203, Chapter 3 *this occurs at* $300\mathsf{m}$ *if* $v_{max} > 120\mathsf{km/h}$*).*
- *There is a structural element allowing a train to send information, for example, a track clearance detection device (axle counter), or the end points of switches that transfer information when passed over.*

We also model the change of gradient on a track as a PIF, as this information is needed to compute breaking distances correctly. PIFs are an abstraction that assume that all these elements have no length, or can be represented by *multiple*

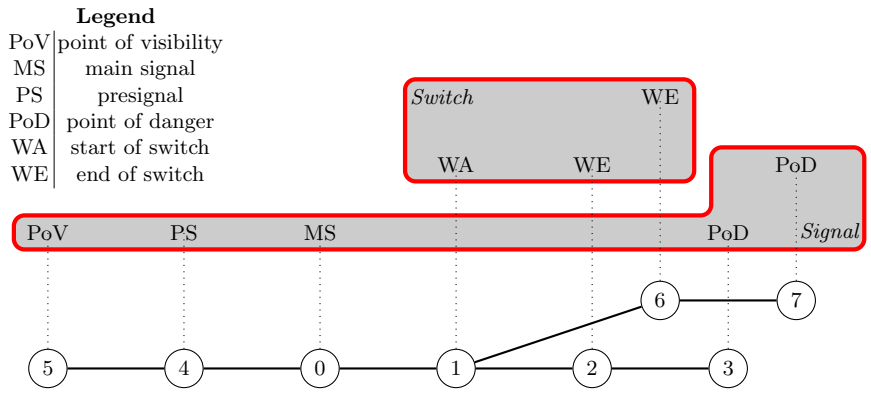


Fig. 1: Structure of a Station Entry

PIFs modeling their beginning and end (for example, switches). This simplification reduces the accuracy of the simulation of physical properties, such as the exact position of a train. Our model, however, is designed for the precise analysis of communications and operational protocols. Information about the exact physical behavior could be obtained from tools for cyber-physical simulations, if so desired.

Each graph node in a rail track plan is modeled as an object of class `NodeImpl` and has a list of objects of subclasses of `TrackElement`. The latter represent different kinds of PIFs and we refer to them as *track elements*.

Track elements are grouped into *logical elements*. For example, a main signal, a presignal, the points of earliest visibility of the presignal, and the points of danger covered by the main signal are grouped as a *Signal*. Fig. 1 shows the entry to a train station with one entry signal and one switch. A signal can have multiple points of danger or visibility and two signals can share one presignal.

We refer to edges between two nodes as *tracks*, to the set of tracks between two signals as *section* and to the set of tracks between the exit signal of one station and the entry signal of another as *line*. The track lengths are modelled as an attribute of edges. There may be multiple lines between two stations.

Nodes, edges, logical and track elements forward information, but do not initialize or delay communication. When a train passes a node, the method `trainLeaves` or `trainEnters` is called (depending on which part of the train passes the node). To model the communication protocol, then either method `triggeredFront` or `triggeredBack` of all track elements on the node is called. Its return value is propagated back to the train and, eventually, the information is also propagated to the station. In this manner a train can read, for example, the state of a signal. Also the station can call state changing methods on all logical elements in its area.

A train only receives information about the current state of the track elements at a node it passes. There is no direct communication between the train and the controller. Similarly, a train only initiates the communication that it passed a node, the station does not receive the identity of the train, only the information that it passed a point of danger. A station, however, knows which trains are in its area and a train knows which station is responsible for it. This is necessary so that a station can issue emergency break orders, etc., and for a train to contact its station in case of a fault. The communication carried out during those situations is carefully separated from regular communication in the model. Neither logical nor track elements advance time.

3.2 Trains

Trains have two positions, front and end, each modeled as the distance on a track relative to the most recent node. For example, if the front of a train is on track e, $5\mathsf{m}$ behind a node n, then this position is described as $(e, n, 5)$. A train has a speed, an acceleration state (stable/braking/accelerating) and a length (the distance between its front and its end) as well as attributes such as maximal acceleration and brake retardation that depend on the production series.

Edges maintain a pointer to the trains that pass them, so if a train occupies more than two edges the information that it occupies the edges in between the first and last is not lost.

Trains are modeled to drive on *simulation events*. At every PIF where the train is active, it computes its next event and the time until this event must be processed. There are three kinds of simulation events:

- The front of a train reaches the next node
- The end of a train reaches the next node
- A train stops accelerating/braking

When a train stops, it does not compute a new event. It can, however, receive a command, directly from the station or by observing a signal, set its state to accelerating and continue driving.

Consider the simulation event when a train reaches a node `n` with its front. It receives information from all track elements at this node and changes its state according to that information. Fig. 2 displays the method to process such an event. Line 7 changes the state to an emergency brake when passing a "Stop" at the main signal, unless *Order 2* (pass the next "Stop" at a signal) was issued.

Not all events are computed by trains. A station can issue an order at any time by calling method `Train.command`. In this case the train computes (i) its current state, based on the current time and the most recent state, (ii) changes its state according to the issued order, (iii) computes, based on the state change, the next event. The process that waits for the old (now invalid order) cannot be canceled in ABS. Instead a counter of the number of orders the train received is increased. When the process of the defunct event is reactivated it checks this counter and immediately terminates if it has advanced.

```
await duration(t,t); //wait
List<Information> li = await n!triggerFront(this, now(), posFront);
while (j < length(li)) {
  Information i = nth(li,j);
  case i {
    Info(STOP) => //passing main signal
      if (!listContains(orders, Ord2)) {accelState = Emergency;}
    StartPrepare(STOP) => //passing presignal
      if (!listContains(orders, Ord2)) {accelState = Break(0);}
    ... //other branches
    _ => skip;
  }
  j = j+1;
}
... //updating location
this.detNext(); //compute next event
```

Fig. 2: Train front arrives at a node `n`

Currently the trains in our model always accelerate and break with maximal force and drive with the maximal permitted speed at each point. In future work we want to model different driving profiles as well as phenomena such as roll out. We expect this to be straightforward.

3.3 Stations

The German railway system has different modes of operation for driving trains outside and inside of stations. Here we focus on operation outside of stations. We differentiate between two kinds of stations: *Blockstellen* which operate block signals and only divide a track line into two parts to increase the possible number of trains on the line and *Zugmeldestellen* (Zmst, simply called "station" for short) which are able to "store" trains and rearrange their sequence. The generalization of both is *Zugfolgestelle* (Zfst).

Each signal is assigned to exactly one Zfst managing it and every switch is assigned to exactly one Zmst. The Zmst is responsible to set the switches and signals correctly when a train passes.

Each Zmst A has a schedule consisting of a list of tuples: time t, train number z, outgoing signal S and target Zmst B (by convention, trains go from A to B). For each schedule item, the Zmst launches a process that waits for t seconds and then attempts to set signal S to "Go" to let z pass. Entry signals are set to "Go" when a train was announced to arrive at this signal, exit signals are set to "Go" when a train is issued to leave on this signal, is accepted and the signal is not locked. To let a train drive from Zmst A to Zmst B on a line L, the following conditions must be fulfilled:

– It is possible to set the signal at A covering the first section S of L to "Go", i.e., S is not locked by A and A has the permit token for S.
– B accepts the train and is notified about its departure.

There are three communication protocols to ensure this:

Locking sections. Each Zfst is responsible for several logical elements such as switches and signals. In addition to the internal state of the signals, the interlocking system itself has a state that depends on the neighboring Zfst. Each section has an additional Boolean state *locked*. Consider a signal covering a section leading out of the Zfst. After a signal is set to "Go" and a train passes it, the section it covers is automatically *locked* and the electronic message "*preblock*" is sent to the subsequent signal. A signal cannot be set to "Go" again, as long as the section it covers is locked. It must be unlocked by receiving the "*backlock*" message from the subsequent signal. That signal in turn can only send "*backlock*" after the train passed. This is one of the measures preventing a track section being occupied by more than one train.

Permit token. For each line there is one token that allows a station to admit trains on this line. Without the token the signal that covers the track cannot be set to "Go". There are various safety protocols to acquire a token. Here we consider the following: To acquire a token, station A must request it from its counterpart B. The request is granted when all trains that left B in direction of A have arrived. Upon initialization the token is given to exactly one station on each line.

Accepting and reporting back trains. Before a train leaves a station A with destination B, A *offers* the train and waits for B to accept. This ensures that B has (or will have) a track to park the train. Before the train departs, the departure is *announced* to B. Once the train arrives, B may report back to A that the train arrives. This is not obligatory in modern systems, as long as no fault occurs. For modeling purposes we assume that all trains are reported back.

The code in the upper part of Fig. 3 shows part of the code modeling the protocol from station A's side: Lines 2–5 ensure that A has the permission to use S. The method `reqPermit` terminates after B granted the request for the token. Line 7 ensures that A does not lose the permit while waiting for B to accept the train, by explicitly forbidding it (allowing it again in 12). Line 8 offers the train to B and line 9 notifies about the impending departure. Line 11 suspends the process until the next section is unlocked. The code in the lower part of Fig. 3 is the method modeling the request for the permit token from B's side: The first conjunct in the guard waits until there are no more trains on S from B to A and the second one waits until B has the token.

Only trains and Zmst advance time, trains by waiting for their next event, Zmst by waiting for the next item in their schedule.

```
// ... extract correct signals and sections
if (!lookupUnsafe(permit, S)) { //Zmst does not have permission
  await nextM!reqPermit(this, S); //acquire permit token
  permit = put(permit, S, True);
}

permitLock = put(permitLock, S, True); //lock token
await nextM!offer(train, this); // offer
nextM!notify(n, lookupUnsafe(duration, nextM), this, A); // register

await !lookupUnsafe(outLocked, S); // wait until next section is free
permitLock = put(permitLock, S, False);
// ... set train as departed and set signal to "Go"
```

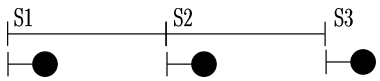
```
Unit reqPermit(TrainNotify sw, Route rtNotify){
  Route rt = getOther(inNotify, rtNotify);
  await lookupUnsafe(expectOut, rt) == Nil &&
        lookupUnsafe(permit, rt);
  permit = put(permit, rt, False);
}
```

Fig. 3: Protocol of the offering station and for releasing the token

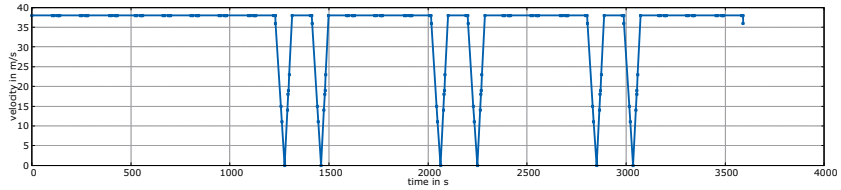
4 Dynamic Analysis

ABS models with initialization blocks are executable and can be compiled into Java 8, Haskell, Maude, ProActive, and Erlang. The concurrency model described in Section 2 is implemented as a runtime environment. In this section we show it can be used to analyze dynamic behavior of a concrete track plan. The object-oriented paradigm of ABS allows to vary the behavior and to perform comparisons between different versions without the need to make global changes to the model.

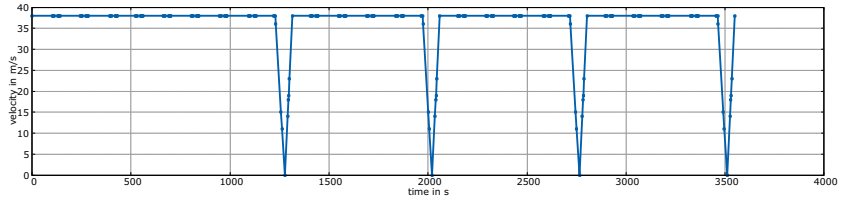
The *Fahrdienstvorschrift* regulates not merely the behavior of trains and stations during normal operation, but also in case of errors and incidents. As an example, we modeled the behavior for the case when a signal cannot be set back to "Stop". In the terminology of safety-critical systems, this would be called a "single stuck-at-Go fault". We describe the scenario with the following diagram:



A train passed signal S2 which cannot be set back to "Stop". As a consequence, S1 cannot be set to "Go". Additional communication and explicit orders are required to mitigate this situation, such that trains may continue using this part of the line. According to *Ril 408.0611* and *Ril 408.0411*, the following communication protocol applies:



(a) Without modification



(b) With modification

Fig. 4: Comparison of train behaviors in case of a faulty signal

1. The train dispatcher T2 responsible for signal S2 communicates to the train dispatcher T1 responsible for signal S1 that signal S2 cannot be set to "Stop".
2. When a train arrives at Signal S1, then T1 requests a *Gleisfreiprüfung* (clearance check) for the track section between S1 and S2, as well as the section between S2 and S3.
3. After clearance is confirmed the train receives two orders:
   Order 2: Pass signal S1, despite S1 signaling "Stop"
   Order 14.4: Stop at signal S2, despite S2 signaling "Go"
4. Once the train arrives at signal S2, T2 issues an *Order 2* to pass signal S2.

The communication protocol has four endpoints (including the train dispatcher responsible for S3 who ensures that the track between S2 and S3 is clear). It cannot be represented and, therefore, is not analyzable in a model that is focussed on a single interlocking station. According to *Ril 408.0411*, the train must always halt before it can receive orders directly from the train dispatcher: *one* broken signal causes *two* stops for each train passing this network section.

The train is always ordered to stop at signal S2, even though it is has been checked that the next section is clear. The reason is that signal S2 might cover a switch. The *Gleisfreiprüfung* only ensures that the section is clear, but not that the switches are set correctly. Hence the train must halt to give the dispatcher an opportunity to set the *Fahrstraße* (train route) correctly.

To optimize capacity one could consider to refine the rulebook such that there are two rules—one for signals covering switches, as described above, and one for signals not covering switches. In the latter, *Order 14.4* in item 3 and item 4 in the communication protocol is not given. Changes in rulebooks incur considerable expenses caused by safety analysis, training, certification, etc. To decide whether this is justified, one has to estimate the expected capacity increase.

Capacity is hard to determine and always requires a concrete track plan and schedule [15]. As a proof of concept for our approach, we modeled a simple track plan with five Zfst arranged in a circle having a circumference of 22.5 km, where one signal has the stuck-at-Go fault described above.[2]

We simulated how one train runs on this track for 3600 s. The resulting v-t diagram is shown in Fig. 4. It can be seen that requiring one stop less decreases delays—the train needs 787 s for a round with the original rule and 744 s for a round with the changed rule, a decrease of 5%.

The original and the changed ABS model are illustrated in Fig. 5: in the method that models a Zfst setting its signal to "Go", we simply issue one order less to the train than before.[3] In this example, the fault itself is deterministic. It is part of the input, which signal breaks at what point in time. The simulation takes less than one second with the Erlang backend of ABS.

The modeled scenario is, of course, a mere approximation of actual railway operations: the train is assumed to always drive with maximal speed and acceleration, the track plan is not realistic. However, it demonstrates that our modeling approach can be employed to analyse the effect of rule changes. In the future we intend to enrich our model with realistic speed parameters, simulating the average behavior of train drivers. As explained in Section 3.1, the track plan in our ABS model is encapsulated in a graph. It is possible to generate this graph automatically from actual track plans available in digital form.

```
1 TrainI train = await s!getObserver();
2 await train!acqStop();
3 train!order(list[Order2, Order144]);
```

```
1 TrainI train = await s!getObserver();
2 await train!acqStop();
3 train!order(list[Order2]);
```

Fig. 5: ABS model of original and alternative rule at Zfst for faulty signal

5 Static Analysis

The EN 50128 [3] standard recommends the usage of formal methods in software development for railway control systems. Our approach is a model on the *architectural level*, i.e., we abstract away from the concrete software and hardware.

[2] Model available at formbar.raillab.de/index.php/en/publications-and-tools/demo

[3] This model transformation is not a behavioral refinement, therefore, it cannot be captured in refinement-based formalisms. ABS offers *software product lines* as an effective method to manage and track changes, see [12] for a detailed discussion.

For distributed software services and, in particular, cloud-based applications, for which ABS was originally developed, the usage of formal methods at the architectural level is established [21]. In this section we argue that railway systems benefit as well from using formal methods at a high level of abstraction.

As pointed out in the previous section, some safety properties can only be established at the global level and cannot be analyzed by local verification of a subsystem. This does not imply that local verification at the implementation level is useless or unnecessary: its results can be imported into an abstract model in the form of guarantees or assertions.

As described at the end of Section 2.3, the strength of ABS's concurrency model is that it allows to decompose global invariants into local ones which are then checked separately for each method. That is possible, because of a rely-guarantee argument where guarantees are justified by strong data encapsulation (all fields are strictly private): this implies that any ABS code between two release points behaves atomically and, hence, can be verified like sequential code. It greatly simplifies reasoning about concurrent systems.

Formal verification rests on *invariants* that are assumed when code is started or resumed and must be established when it suspends or terminates, in other words, they must hold whenever communication takes place in the modeled system. In concrete terms, each method is proven separately to preserve its local class invariant. It is not necessary to explore the global state space of a system and invariants are established without reference to an initial state. In the railway context this means we are able to reason about behavior *without a concrete track plan.* As an example we consider the following property:

> "Let S be a section between two Zmst A, B. If A releases the permit token for S, then there are no trains on S in the direction of B." (1)

This means that, if B requests the token and A releases it, then all trains in the direction from A to B have already arrived in B.

Depending on the interlocking systems in the station, different mechanisms to ensure this property are in place. Here we consider a variant of an older interlocking system, where the permit token is not secured technically, but transferred by a phone call between train dispatchers. To transfer the token, the dispatcher of that station which currently does not have it calls his counterpart and requests transfer. The other dispatcher may only release the token when all trains that departed from his side have been reported back.

In this paper we present our modeling approach and provide a proof-of-concept, hence a full-fledged case study that includes verification of the complete interaction between nodes, track elements, logical elements, Zmst and trains, is out of scope. In particular, we make some assumptions:

A.1 Lines are encoded correctly, i.e., a line L from A to B is encoded with its first section on A and its last on B and there are tracks that connect A and B using the correct in- and outsignal.

A.2 Tracks have length strictly greater than 0.

Property (1) can be expressed as a history invariant which is formalized in first-order logic and can be verified with KeY-ABS [7]. In the following formula let A, B be two Zmst and S, $\hat{S}$ two sections of a line L such hat S is the first section of L from A and $\hat{S}$ the first section of L from B. It expresses that when A releases the permit token to B, every train that was announced from A to B was reported back by B to A.

$$\begin{aligned}
&\forall\,\mathsf{Int}\ i;\ \big(\exists\,\mathsf{Fut}\ f;\ h[i] \doteq \mathsf{futEv}(A, f, \mathsf{reqErlaubnis}, (B, S))\big) \rightarrow \\
&\quad \forall\,\mathsf{Train}\ T,\ \mathsf{Int}\ j;\ j < i \rightarrow \\
&\qquad \Big(\big(\exists\,\mathsf{Fut}\ F;\ h[j] \doteq \mathsf{invREv}(A, B, f, \mathsf{anmelden}, (T, t, A, S))\big) \rightarrow \\
&\qquad\quad \exists\,\mathsf{Int}\ k,\ \mathsf{Fut}\ f;\ h[k] \doteq \mathsf{invREv}(B, A, f, \mathsf{rueckmeldung}, (B, T, \hat{S})) \wedge j < k < i\Big)
\end{aligned} \tag{2}$$

Theorem 1. *Invariant* (2) *holds for method* `reqErlaubnis` *in Fig. 3 (and all other methods in its class), i.e., if it holds at the start of the method, then it is reestablished after termination.*

This does not yet show that there are never two trains on one line in opposing directions. To show that one must additionally establish that if a train enters a line, then it was offered, accepted and announced and that when a train is reported back, then the train left the line. A proof sketch of Theorem 1 is in the Appendix. It has also been proven mechanically with the help of KeY-ABS.

6 Related Work

The work closest to ours is by James et al. [17], who presented a formalization of ETCS level 2 in Real-Time Maude and analyze the communication between trains and one station. Like ours, their approach is set at the design level and encompasses all components needed for driving trains. However, it is restricted to one specific rail yard, necessitated by the use of model checking instead of deductive invariant reasoning. A further difference is that our work concentrates on ETCS level 1LS, which is the most relevant within the network of Deutsche Bahn AG. Maude is an object-oriented language based on term rewriting and one of the backends supported by ABS. Therefore, potentially both modeling approaches might be combined.

Individual rail yard components such as interlocking systems have been analyzed by multiple approaches, for example, recently in SystemC [13], OCRA [19] and CSP||B [20]. An overview over approaches for interlocking systems, the most frequently analyzed component, can be found in the survey of Fantechi et al. [9] and a comparison of ABS with these approaches in [12].

There are two main approaches to combine micro- and macroscopic models:

- Relating several models of increasing abstraction level and using the appropriate one for a given use case. This is either done by generating more abstract models on demand from a microscopic model or annotating the relation between a micro- and a macroscopic model. [4, 16]

– Mesoscopic modeling, which aims to be a middle ground in terms of abstraction, tailored for a given use case. A recent application of this approach was generating timetables by de Fabris et al. [5].

Our approach leans towards mesoscopic modeling, but achieves simplification not by summarizing multiple elements, but by abstracting from certain aspects. For example, we do model each magnet of the train protection system PZB, but assume these as having no length. Similarly, established mesoscopic models do not consider the communication layer, which is the main focus of our work.

7 Conclusion & Future Work

We presented an approach to modeling and analysis of railway systems based on an object-oriented, concurrent, executable modeling language. The modeling formalism is able to unify aspects from micro- and macroscopic modeling and allows to analyze static (for example, safety) as well as dynamic (for example, delays) properties of a rail yard based on a single model. For static analysis we use deductive invariant reasoning which allows to prove properties for any valid track plan and initial configuration.

As the next step, we plan to calibrate and validate our model with real data on a part of the actual railway network of Deutsche Bahn AG. This includes establishing realistic driving profiles regaring acceleration and speed as well as to determine the precision of our approach in terms of train positions. On the safety side, we plan to provide a formalization of all incident scenarios described in the rulebooks [6] and to prove a suitable safety property for this model. Furthermore, we plan to use analysis tools developed for ABS *software* models, such as complexity and deadlock analysis [1, 10], to examine the properties of the rulebook and for carrying out a capacity analysis.

Acknowledgements. We thank Sebastian Schön for his insights into train operations and the anonymous reviewers for helpful comments. This work is supported by `FormbaR`, 'Formalisierung von betrieblichen und anderen Regelwerken', part of AG Signalling/DB RailLab in the Innovation Alliance of Deutsche Bahn AG and TU Darmstadt.

References

1. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: static analyzer for concurrent objects. In E. Ábrahám and K. Havelund, editors, *Proc. TACAS*, volume 8413 of *LNCS*. Springer, 2014.
2. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. User-defined schedulers for real-time concurrent objects. *ISSE*, 9(1):29–43, 2013.
3. CENELEC. DIN EN 50128:2011, Railway applications – Communication, Signalling and Processing Signals.
4. Y. Cui and U. Martin. Multi-scale simulation in railway planning and operation. *Promet Traffic&Transportation*, 23(6):511–517, 2011.

5. S. de Fabris, G. Longo, G. Medeossi, and R. Pesenti. Automatic generation of railway timetables based on a mesoscopic infrastructure model. *Journal of Rail Transport Planning & Management*, 4(12):2–13, 2014.
6. Deutsche Bahn Netz AG, Frankfurt, Germany. Fahrdienstvorschrift Richtlinie 408. August 2016: fahrweg.dbnetze.com/fahrweg-de/nutzungsbedingungen/regelwerke/betriebl_technisch/eiu_interne_regeln_ril_408.html.
7. C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In A. P. Felty and A. Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 517–526. Springer, 2015.
8. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.
9. A. Fantechi, F. Flammini, and S. Gnesi. Formal methods for railway control systems. *STTT*, 16(6):643–646, 2014.
10. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in `core abs`. *Software & Systems Modeling*, 15(4):1013–1048, 2016.
11. R. Hähnle. The abstract behavioral specification language: A tutorial introduction. In E. Giachino, R. Hähnle, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. Formal Methods for Component-Based Systems FMCO*, pages 1–37, 2012.
12. R. Hähnle and R. Muschevici. Towards incremental validation of railway systems. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 7th International Symposium (ISoLA), Part II, Corfu, Greece*, volume 9953 of *LNCS*, pages 433–446. Springer, Oct. 2016.
13. A. E. Haxthausen, J. Peleska, and S. Kinder. A formal approach for the construction and verification of railway control systems. *Formal Aspects of Computing*, 23(2):191–219, 2011.
14. C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In N. J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973.
15. International Union of Railways (UIC). Capacity (UIC code 406), 2004.
16. International Union of Railways (UIC). IRS 30100 - RailTopoModel - Railway Infrastructuretopological Model, 2016.
17. P. James, A. Lawrence, M. Roggenbach, and M. Seisenberger. Towards safety analysis of ERTMS/ETCS level 2 in Real-Time Maude. In C. Artho and P. Ölveczky, editors, *Formal Techniques for Safety-Critical Systems FTSCS, Revised Selected Papers*, volume 596 of *CCIS*, pages 103–120. Springer, 2015.
18. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. Formal Methods for Components and Objects FMCO*, volume 6957 of *LNCS*. Springer, 2010.
19. C. Limbrée, Q. Cappart, C. Pecheur, and S. Tonetta. Verification of railway interlocking, compositional approach with OCRA. In T. Lecomte, R. Pinger, and A. Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification, RSSRail*, volume 9707 of *LNCS*, pages 134–149. Springer, 2016.
20. F. Møller, H. N. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. Defining and model checking abstractions of complex railway models using CSP||B. In *Hardware and Software: Verification and Testing HVC, Revised Selected Papers*, volume 7857 of *LNCS*, pages 193–208. Springer, 2012.
21. C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon web services uses formal methods. *CACM*, 58(4):66–73, 2015.