Formal Modeling and Analysis of Railway Operations with Active Objects

Eduard Kamburjan*, Reiner Hähnle

Software Engineering Group, Technische Universität Darmstadt, Germany

Sebastian Schön

Railway Engineering Group, Technische Universität Darmstadt, Germany

Abstract

We present a comprehensive model of railway operations written in the active object language ABS. 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 and propagation of 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 [1]. 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 stations with the goal of being precise enough to examine fine-grained, local 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 formal models concentrate on one single aspect, e.g., the safety of interlocking and signaling systems or the network throughput.

Railways are complex systems with global properties such as safety or capacity. These depend not only on low-level, structural components, but also on 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 in the whole network. Such failures 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 modeling effort 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.

*Corresponding author

Email addresses: kamburjan@cs.tu-darmstadt.de (Eduard Kamburjan), haehnle@cs.tu-darmstadt.de (Reiner Hähnle), schoen@verkehr.tu-darmstadt.de (Sebastian Schön)

3. Hence, it is possible to analyze the effects that perturbations at a low abstraction level have on the global, system-wide behavior.

In the following, we describe the regulations of railway operations in Germany, as laid down in a variety of laws, rulebooks and technical documentation. We illustrate the current workflow around editing rulebooks and we identify the need for tool support based on a series of interviews with rulebook managers and authors at Deutsche Bahn AG.

Our modeling method uses the active object language ABS [2], 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 operations, 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, as well as others trains synchronizing with this train. 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 has a global impact which is observed at a macroscopic modeling level.

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 opposite directions. We analyze the *communication structure* between trains, infrastructure and stations with *deductive* invariant reasoning, not with model checking, 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 those. We concentrate on procedures and communication, and how a fault is handled *at 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 ensures safety. Such procedures are described in detail in the *Fahrdienstvorschrift* [3] for all railways in Germany operated by Deutsche Bahn AG. Our model is a partial formalization of the description of ETCS L1 LS within that rulebook. Our main contributions are:

1. A novel, uniform modeling approach of railway operations in the concurrent, executable language ABS that allows static and dynamic analysis.

2. A deductive invariant-based analysis of safety of railway communication.

Outline. The paper is organized as follows: In Section 2 we describe the current workflow for rulebooks in railway operations. In Section 3 we present the ABS language and in Section 4 our model of railway operations. In Section 5 we show how changes in procedures can be analyzed by simulation. In Section 6 we show a safety property and show how ABS admits its formal proof. Related work is in Section 7 and we conclude in Section 8.

Relation of this paper to previous work. This article extends [4]. Section 2 is completely new, there is an extended presentation of ABS and the ABS Dynamic Logic in Section 3, and substantial extensions on the modeling in Section 4, which now covers all concepts needed to describe ETCS L1 LS in detail and was partly presented in [5]: the guiding example and all implementation details are new, in particular the internal state description of trains and stations, the event generators and the discussion on faults and driving without intermediate PIFs. The case study in Section 5 is new and more complex, featuring multiple trains to show propagation of delays. The proof of Theorem 6.1 is provided in the appendix.

2. Railway Operation Regulation Process

2.1. Regulations

In the German railway system, railway operations are regulated by laws and regulations issued by the federal government, as well as rulebooks written by the corporation managing the infrastructure. These rulebooks are partially derived from the laws and regulations and implement procedures to ensure the adherence to the law. But they also implement details not demanded by the law and cover aspects only indirectly related to operations. Each rulebook is structured as a set of definitions and modules encapsulating single aspects and situations. Additionally, implementation details of procedures are described in internal technical documents. The regulations related to operations can thus be grouped by their purpose into three categories:

Planning and building of railway infrastructure. For the considered mode of operations, largely identical to ETCS L1 LS, this corresponds to rulebook *Ril 819* [6], where guidelines for the planning of track layout and track parameters are described, e.g., minimal and maximal distances between pre- and main signal, or distances for visibility. Signal aspects are described in the *Eisenbahn-Signalordnung* [7] (*Law for Railway Signals*), while interlocking implementation is described in training material [8] and internal documents.

Planning of railway operations. DB Netz AG issues rulebooks for proprietary use on how to conduct feasibility studies and timetable studies based on the concept of operations of a railway undertaking. These regulations were not relevant for this work.

Rulebooks for railway operations. DB Netz AG issues two versions of the main rulebook for railway operations *Ril 408*: One version for staff of the infrastructure manager, e.g., the operator of an interlocking system. The second version for the staff of the railway undertaking, e.g., the train driver. This rulebook includes all rules of operation for trains following train signals (conventional and ETCS). *Ril 408* [3] describes all rules of operations step-by-step during normal operations and during deviations on an abstract level. Specific rules, e.g., for a certain type of interlocking system, can be found in the set of rules in *Ril 482* [9]. These books reference the rules for infrastructure, mainly *Ril 819*. Relevant laws for the operations are described in the *Eisenbahn-Bau- und Betriebsordnung* [10] (*Law for Operating and Building Railways*).

Prior to choosing our modeling approach, we conducted a scoping study with seven rulebook managers of DB Netz AG, the biggest German infrastructure manager, to identify where tool support is needed. The interviewees were responsible for rulebooks related to operations or infrastructure, or managing rulebook changes.

2.2. Workflow for Changes

Changes are requested either by changing laws, reactions to incidents, feedback from trainers (especially regarding ambiguous rules), the need for local deviations, or the introduction of new operation modes like ETCS. Change requests are bundled by the responsible manager of a rulebook and delegated to the responsible rulebook authors. Detailed knowledge about the contents and on cross-references to other rulebooks that is required for rulebook maintenance is handled by a group of authors or a single author. Overarching knowledge is shared during workshops and by informing authors of related rules about changes. The documentation of all changes is public (to the group of the readership) and added to any re-issue of a rulebook as a supplement.

2.3. Authors' View

Rulebook authors are supported by software tools for change management and by software tools that simulate physical aspects of railway operations, including microscopic simulation of single trains (e.g., to compute braking curves). More complex simulations may be outsourced to research institutes. Macroscopic tools are not used during rulebook changes, but are established in other departments. We identified several observations related to rulebooks which were mentioned by multiple interviewees:

1. The module structure helps in distributing the work of rulebook authors, but makes it difficult to obtain a global view on non-trivial operation scenarios. This makes changes that range over multiple responsible authors time-intensive.

2. On several instances, the ambiguity of natural language made clarifications of a rulebook necessary. A formal and easily communicable form of describing railway operations was asked for.

3. The rulebook authors are also responsible to approve local deviations from the regulations by ensuring that these deviations guarantee the same level of safety. While physical properties can be analyzed precisely with automatic tools, the analysis of logical properties is purely manual.

The uniform model presented here aims at providing a formal model that can serve as the basis of prospective tools that support rulebook authors and other contributors to railway regulations. In particular, we intend to address the following issues: The formal model has no ambiguity, it offers a global view, yet is flexible enough to analyze safety and capacity effects of local changes. Even though the model is derived from German rail regulations, the approach presented here is applicable to other national and international operation frameworks, because all of these regulations are very similar to ETCS L1 LS.

3. Modeling with the ABS Language

The abstract behavioral specification (ABS) language is an object-oriented, executable modeling language designed to model and verify concurrent and distributed software systems [2]. To cope with the difficulties of verifying complex, distributed systems, the design of ABS embodies two main abstraction principles to reduce complexity: a concurrency model that permits compositional verification and an explicit model of product variability. The ABS concurrency model extends the actor [11] paradigm as follows:

- Objects on different processors do not share memory and all ABS objects are strictly encapsulated and have neither public nor static fields. Communication between objects is only possible via *asynchronous method calls*. Thus writing and reading of an object's memory is always under control of that object. If another object must access its memory location, it must do so with getter and setter methods.

- At every moment, at most one process is active on a given object. The object's processor may not change the active process at any time, instead methods define explicit synchronization points at which such a change may occur.

Together, these properties imply that between two explicit synchronization points a method can be regarded as sequential and, therefore, it can be analyzed with theories developed for sequential programs.

In ABS variability modeling and code reuse is not achieved by class inheritance, but by traits and deltas [12]. Classes may not extend other classes, hence dynamic dispatch does not occur and it can be determined statically which code will be executed at a call point.

The ABS syntax is loosely based on Java and most concepts of ABS are (intentionally, to ease its usage) standard. Below we introduce the language with a 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 how to perform formal verification. A comprehensive account of ABS is given in [2, 13].

ABS models can be compiled into executable Erlang, Java, Maude, ProActive and Haskell code. In this case an *initialization block* must be provided (which is unnecessary for deductive static analysis). This is a special ABS statement that serves as the entry point to execution of a model. While ABS classes describe general behavior, the initialization block sets up a concrete scenario.

Compilation into executable languages allows to use ABS not solely for modeling and static verification, but also to run the models and verify properties dynamically by analyzing single generated traces. Section 5 shows how dynamic analysis through execution is used to determine runtime of trains on fixed track layouts. The main tool for static verification, the ABSDL logic, is introduced in Section 3.5 and an example for verification is given in Section 6.

3.1. Functional Model

Because of the strong encapsulation of ABS objects, it is not viable to model every concept in a purely object-oriented manner. To model *immutable* data in a concise way, ABS uses *algebraic data types* (ADT) as well as built-in types like integers, rationals, strings, lists and maps that are implemented as ADTs. Elements of such types can be passed around and stored in the heap memory, but have no own memory or processor. They may refer to objects (i.e., have a parameter typed with an interface).

Example 3.1. *The state of a signal and the acceleration state of a train are modeled as ADTs:*

```
1 data State = GO | HALT | SLOW | INVALID | NOSIG;
2 data AccelState  = Brake(Rat target) | SpeedUp(Rat target) | Emergency | Stable;
```

Computations and operations on ADTs are defined in a functional expression language that is part of ABS. This sublanguage includes the usual operations for arithmetic and list processing, but no higher-order functions. Both functions and ADTs may be parametric in types. To process user-defined ADTs, functions may pattern-match on the type structure.

Example 3.2. *The following function takes a list of pairs as an input and returns the second element of the first pair whose first element matches the key:*

```
1 def T2 find(List<Pair<T1,T2>> list, T1 key) =
2   case list { Cons(Pair(a,b), xs) => if(a == key) b else find(xs, key); };
```

This function is not side-effect-free: if the list contains no such pair (e.g., when it is empty), the end of the list will fail to pattern-match and an exception is raised. Implicit exceptions are the *only* allowed side effects in the functional language, neither method calls nor field accesses are permitted.

3.2. Object Model

ABS is an object-oriented language with a strict *programming to interfaces* discipline: Only interfaces are exposed to clients and only interfaces constitute reference types. Classes may implement multiple interfaces and interfaces may extend other interfaces, but there is no class inheritance.

The syntax of interface and class declarations is very close to Java syntax. Instead of a constructor, classes have class parameters and an initialization block that is executed at object creation. Additionally, classes may have a recovery block: ABS uses a Java-like exception model. An exception is either raised implicitly, e.g. by division by zero, or explicitly by the `throw` statement. If the exception is not caught, the process terminates and the recovery block is executed.

Example 3.3. *The following code defines a client that connects to a given server after creation and offers to return its id. Method* `m()` *is only callable by the object itself, as it is not exposed outside the interface.*

```
interface Client {
    Int requestId();
}
class Client(Int id, Server s) implements Client {
    {                                // init block
        s!connect(this);
        this!m();
    }

    recover {                        // recovery block
       _ => s!logError(id);
    }

    Int requestId() { return id;}    // interface method

    Unit m() { ... }                 // internal method
}
```

ABS allows to advance time explicitly in processes [14]. The statement "`duration(t1,t2);`" blocks the active process between `t1` and `t2` time units. ABS leaves open how long a time unit is (in this paper we use seconds). The local time can be accessed with `now()`.

Most ABS statements are standard and very similar to Java, with the main difference that field accesses can only refer to the fields of the same object, not to fields of other objects of the same class—encapsulation of ABS is more restrictive than **private** fields in Java. The additional concepts of ABS are centered around communication and time and we present the corresponding statements in the next section.

3.3. Concurrency Model

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 3.4. *The following code calls method* `m()` *on the object stored in* `o` *and saves the future in the local variable* `f` *(line 1); it waits for* `m()` *to terminate (line 3) and stores the return value in 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 o!m();`*".*

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 (it also releases control if `g` already holds initially). 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;

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

- 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 below.

Explicit release of control reduces the number of possible interleavings between processes that are possible as compared to preemptive scheduling, since between the **await** statements, a process has exclusive control over the object memory and can be regarded as sequential.

If a future is accessed before its process has terminated, the active process blocks and the object blocks as a whole until the process of the future has terminated, for example, "`Fut<Int> f = o!m(); Int i = f.get;`". There is another shorthand notation for this: "`Int i = o.m();`".

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

The time advancing statement `duration` can also be used as a guard expression and advances time while suspending the active process: "`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 without advancing its clock.

Objects can be grouped into *concurrent object groups (COGs)*. Within a COG, the heap memories of the included objects are still separate, but the processor is shared. This means that only one object of a COG can be active and execute one of its processes.

3.4. Variability and Code Reuse

ABS uses *delta-oriented product lines* [15] for variability management. A product line does not consist of merely one ABS model, but a whole set of them, so-called product variants that differ in the realized features. For example, one feature might be normal operation and another feature then could be operation under the assumption of a specific fault, such as a broken signal.

Since all product variants implement the same base functionality, it is natural to structure the implementation artifacts of a product line into into (i) *core* code that is common to all products and (ii) implementation *deltas* that specify how the core code has to be modified in order to generate a product variant that implements a given feature. A product variant is thus defined by a set of deltas and an application order.

Technically, a code delta defines transformation operations that add, modify or remove classes, interfaces and ADTs. Their application is syntactically defined (plus some type checking [16]) and thus rather lightweight. In our modeling approach we use deltas to manage multiple model variants; in particular, we use them to add faults and fault handling.

Deltas are intended for code reuse across different product variants. As ABS objects have no code inheritance, a different mechanism for intra-product code reuse is needed. Here we use traits [17], i.e. methods that are defined outside of a class context and can thus be imported at any location. ABS traits can also be used inside code deltas. In fact, ABS traits and deltas share a uniform syntax [12]. Traits and deltas are not the focus of the present paper. For an extended discussion of the usage of deltas in railway modeling we refer to [18].

3.5. Four Event Semantics

The formal semantics of ABS can be described with the help of communication events, each describing a communication action of a future [19]. We use four different events, one for each possible action of a process on a future 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 3.1 (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:*

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

An invocation event is added when the object 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 moment in time as the invocation, but not that it is immediately executed. As there are no assumptions about the scheduler, this means that there is no guarantee that two methods will be started in the same order as they were called. The completion event is added once the process which has f as its handle terminates with the return value e in object o'. The completion 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 in a method call or be stored in a field in the heap. Figure 1 illustrates the events in a sequence diagram.

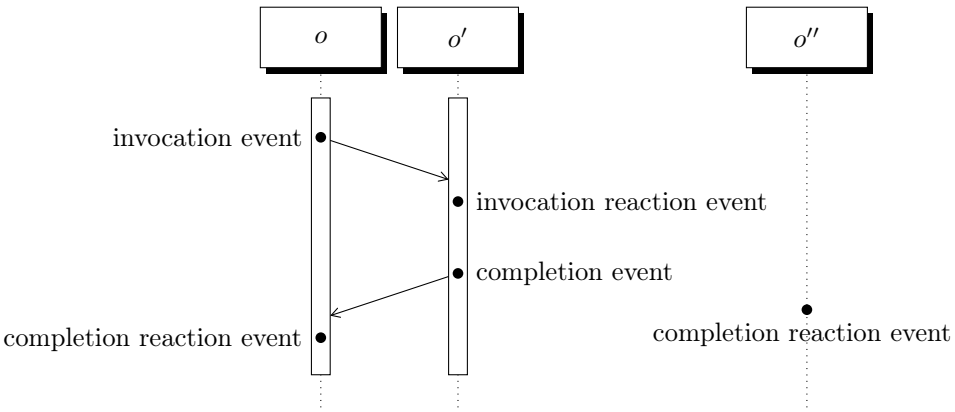


Figure 1: Events on futures, diagram taken from [20]

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 $h[i] \doteq \mathsf{invREv}(o, o', f, m, e^*)$, then there must be a $j < i$ with $h[j] \doteq \mathsf{invEv}(o, o', f, m, e^*)$. This condition expresses that every process starts its execution only after it was called. The well-formedness conditions for all event types are listed in [19].

Histories are axiomatized as a theory of finite sequences. We can express invariant properties over histories as formulas in first-order logic. 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.

Example 3.5. *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:*

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

Because of strong encapsulation, however, 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 [20] is then able to statically and formally verify that each method in a class preserves its local invariant.

To do so, a formula modeling the preservation of the invariant in *ABS Dynamic Logic (ABSDL)*, a program logic for ABS, is proven to be valid for each method. ABSDL axiomatizes a modality over ABS statements, the first-order theory of histories sketched above, as well as additional theories for heap memory and arithmetic.

Example 3.6. *Let* $I \equiv \mathsf{select}(\mathit{heap}, \mathit{self}, i) \geq 0$ *(where the heap access function* $\mathsf{select}(\mathit{heap}, \mathit{self}, i)$ *models the object access* `this.i`*) be an invariant and "*`this.i = 0; await this.i > 0; other!change(i);`*" the code in the method body of* `m()`*. The proof obligation to show that* I *is an invariant for* `m()` *is:*

$$I \rightarrow [\mathbf{this}.\mathtt{i} = 0;\ \mathbf{await}\ \mathbf{this}.\mathtt{i} > 0;\ \mathtt{other!change(i);}]\, I$$

It says that if I *holds at the beginning of the execution of* `m()`*, then* I *holds* at every point the process releases control*, i.e. when it terminates and at its suspension in the second statement.*

With invariant-based reasoning we are able to state and prove properties over *all* histories realized by a system, while each execution of the ABS model generates only one history. However, the four event semantics in [19] does not include the timed semantics of ABS yet and is thus not able to express properties concerning time.

3.6. Execution and Analysis of ABS Models

The ABS language has been specifically developed with the aim of being executable as well as analyzable [2, 13]. In fact, the results reported in the present paper provide constructive evidence for this claim. The ABS tool chain is explained in detail in [21], here we concentrate on the features relevant for the present text.

3.6.1. Model Execution and Dynamic Analysis

The ABS language does not have a dedicated runtime environment, but is implemented by cross compilation to various language backends. At the moment, these comprise Maude, Erlang, Haskell, and Java. The main steps in the compilation process are as follows: The modeler creates an ABS model, including delta modules, traits, and a feature model (see Section 3.4), in his or her favorite editor (support for Emacs, Eclipse, and a web-based interface is available). This is parsed into an abstract syntax tree (AST). The AST is then "flattened" into a so-called Core ABS model, where deltas have been applied and traits inlined. The Core AST is then type checked (an experimental type checker working on full ABS is available, see [16]). Code generators for the various backends work on the type checked Core AST. Since ABS is type safe, it is not necessary to include type checks in the generated code. Code generation can be triggered either from an IDE with ABS support (Emacs, Eclipse, Collaboratory) or from the command line, with a command such as "`absc -erlang model.abs`". The resulting program, possibly with some ABS-specific libraries, is then executed or compiled in each backend as usual.

The Maude backend implements directly the SOS semantics of Core ABS [2]. It is rather slow, uses a lot of memory, and is mainly used when developing new ABS language features. For the work reported in

this paper we used the Erlang backend and various shell scripts. Information on the currently supported ABS backends can be found in the ABS Language Manual [22].

Regarding the analysis of ABS models we distinguish between *dynamic* and *static* analysis. Dynamic analysis essentially consists of running different scenarios on complete, executable ABS models that must have an initialization block (see introduction to Section 3) that contains one such scenario. In many cases, visualization tools are used to focus on the relevant aspects of a run, see Fig. 14. Dynamic analysis is essential to validate and calibrate an ABS model. But its main purpose is to execute scenarios that would be difficult or expensive to try on the modeled system. For example, dynamic analysis can be used predict scalability of the modeled system [23] or, as done in Section 5, the performance impact deriving from variations in the modeled system.

3.6.2. Static Analysis

The ABS ecosystem contains a number of static analysis tools. These implement various complex analyses of semantic properties. All of them rely crucially on the ABS concurrency model that is based on asynchronous method calls with explicit futures and release points, see Section 3.3. This makes it possible to analyze the code between two release points in the same manner as sequential code and so drastically reduces the number of possible interleavings when compared to a language with preemption. ABS features automatic deadlock analysis [24] and cost analysis [25], both of which work on safe abstractions of ABS programs, i.e. they are sound, but not always precise.

In addition, it is possible to automatically generate systematic test cases from a given ABS program [26] and to formally prove properties by deductive verification [20]. Both analyses are precise, but test generation is obviously not exhaustive. In the following, we concentrate on the latter, because it is used in Section 6.

Assume that all methods declared in a COG maintain the same invariant I, i.e. whenever I holds at the start of the execution of an object or when execution is resumed, then it is also guaranteed when execution finishes or is suspended. In ABS all these release points are known. Hence, one can establish that I holds at all of these points by deductive verification. Due to object encapsulation, only a (setter) method of a COG can change the state of that COG. But these methods must preserve I as well. In summary, if all methods of a COG preserve the invariant I at each release point locally, plus I is established by the constructors, then I must hold any release and resumption point for *any sequence of methods calls* [19]. In other words, ABS allows *compositional* invariant reasoning: any invariant shown locally for each method is also an invariant of the whole COG.

Our invariants take the form of first-order properties over events and program data as explained in Section 3.5. Compositionality lets one break down the proof of an invariant for a COG to a set of proofs that establish it for each method. Such a local proof validates essentially a Hoare-triple of the form $\{I\}\mathtt{p}\{I\}$, where I is the invariant to be proven and `p` is an ABS program without suspension points. This kind of judgment can be proven with a deductive verification tool for *sequential* programs. There is a variant of the deductive verification tool KeY [27] that can prove such statements for the event-based specification language described in Section 3.5 and for sequential Core ABS programs [20].

4. The Railway Operation Model

4.1. Assumptions

As described in Section 2, our model is focused on operations described in rulebooks. Not all components are described in these rulebooks—their specifications can be found in requirement specifications or technical documents. For instance, the communication between stations is in part described in *Ril 408* [3] 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 the extent they are 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 do not model how trains roll back a short distance after releasing their brakes when starting at a slope.

Our model[2] assumes *instantaneous communication* and *maximal progress*—communication has no delay and is processed immediately, state changes take no time and trains always accelerate and brake with maximal force and drive with the maximal permitted speed. Modeling delays in communication and non-optimal speed will be considered in the future.

4.2. Overview

We model the infrastructure, i.e. track elements, as a layered model centered around *points of information flow*. A point of information flow is a position on the track, where information from or to the track may be transmitted.

Definition 4.1 (Point of Information Flow)**.** *A* point of information flow *(PIF) is a* position at a fixed position on a track, that is used for information flow *where one of the following criteria applies:*

- *It is an infrastructure 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, where the signal is seen at the latest. For example, according to* Ril 819.0203, Chapter 3*, this occurs at* 300m *if* $v_{max} > 120\mathsf{km/h}$ *for pre-signals.*
- *It is an infrastructure element allowing a train to send information, for example, a track clearance detection device (axle counter), or the endpoints of switches that transfer information when passed over.*

We also model attributes of the track, e.g., its gradient, as PIFs, which transmit the information that this attribute has changed. This information is needed to compute braking distances correctly, and modeling track attributes as special nodes simplifies the model, because edges only carry their length as an attribute. The change of such an attribute can be seen as information flow at a structural element. PIFs are an abstraction that assumes that all these elements have no length, or can be represented by *multiple* PIFs modeling their beginning and end.

For example, switches are modeled as *three* PIFs, one for each end. By modeling properties of edges as properties of their limiting nodes, we need not consider edges for information flow and can concentrate on the nodes. This does not result in unnecessary duplication of saved information: While each gradient is saved twice (at the limiting nodes), one does not need to save the direction of the gradient, as this information is implicit depending on the entering node.

The abstraction embodied in PIFs 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 communication and operational protocols. Information about the exact *physical* behavior could be obtained from tools for cyber-physical simulations, if so desired. As mentioned above, a PIF is a position at a track and an object that describes the information to be transmitted or relayed. Instead of modeling all features of a PIF in one object, we use a model of four layers to organize and separate its structure, see also Figure 5:

1. **Topology Layer** This layer is a directed graph, where the nodes model the position of PIFs. A train is modeled as two points (for front and back) with respect to this layer.
2. **Physical Element Layer** This layer models physical elements and the actual device for information transmission at the PIF. Each such physical element has a position, modeled as a reference to the topological layer.

[2]The model is downloadable at formbar.raillab.de/index.php/en/publications-and-tools/demo.

3. **Logical Element Layer** This layer groups physical elements. For example, a pre-signal, a main signal, three ATP magnets, one visibility point for the pre-signal and at least one axle counter (at the guarded danger point) are grouped into a single *logical* signal.

4. **Interlocking Layer** This layer groups logical elements within a station. All communication between physical/logical elements assigned to different stations must go through this layer.

Each layer communicates only with its direct successor and predecessor layers. Hence, the interlocking layer only communicates with logical elements, not directly with physical elements, etc. The layer model facilitates reasoning about behavior as described by documents for the signaler's side (formulated in terms of logical elements) and behavior as described by documents for the train driver's side (formulated in terms of physical elements). The part of the main operations rulebook *Ril 408* concerning faults is defined in terms of both kinds of elements: The rules for the train driver are defined in terms of physical elements, the rules for train dispatcher are (mostly) defined in terms of logical elements. Technical documents are defined in terms of physical elements.

The separation into nodes, physical elements, and logical elements also leads to a clearer model, as it realizes a separation of concerns: nodes serve as interfaces for communication with the train and mark a *geographical position*, physical elements describe the state of a track element, and logical elements serve as an interface to the interlocking system. It also clarifies communication: In the lower three layers, there is no communication between elements on the same layer and no logic: All decisions are made at the ILS layer, the logical elements distribute the information down to the physical elements and these adjust their state. The nodes forward information from the train to the ILS through the other two layers or request information stored in the physical elements and send it to the train.

Our model combines microscopic and macroscopic modeling: The lower three layers are a microscopic model of infrastructure, while the interlocking layer is a macroscopic model of the network. The first three layers are *not* at different levels of abstraction, instead they separate multiple *aspects* of an infrastructure element: The topology layer is modeling the position, the physical element layer the stored information and communication to/from the train and the logical element groups physical elements and models information to/from the interlocking system.

We now describe the layers and their implementation in detail. As an example, we show the elements needed for a self checking speed restricting signal ("Geschwindigkeitsprüfabschnitt vor dem Geschwindigkeitsanzeiger"), short *speed limiter*: The speed limiter works as follows: when the train passes the speed limiter signal, a magnet is triggered and activates a second magnet after a certain distance. The second magnet forces a passing train to trigger an emergency brake. After $\frac{\text{limit}}{\text{distance}}$ seconds, the second magnet is deactivated. Hence, only if a train is too fast it is forced to brake, because it arrives at the second magnet too soon. The speed limiter also has a speed limiter pre-signal, which announces the limit before the speed limiter signal: the train must start braking at the pre-signal, but can only start to accelerate at a main signal.

4.3. Topology Layer

The first layer is an undirected graph, where edges correspond to tracks and are labeled with length and nodes correspond to the *position* on a track of a point of information flow. Figure 2 shows an excerpt from the interface and class for nodes. The figure shows three methods visible to the outside: `getOut` returns the edge opposite to the input edge `e`. Method `triggerFront` returns a list of information items when a train passes this node with its front. The information is not saved or managed by the node, instead the implementing class `NodeImpl` maintains a list `belongs` of references to physical elements on the next layer. The `edge` passed to the method is the direction the train is coming from, as not all track side elements are visible from both directions. The method `triggerBack` is analogous. Information items are modeled as an ADT `Info`, which contains (among others) the following definitions:

```
1 data Info =  NoInfo | Info(State) | Prepare(State) | StartPrepare(State)
2   | Limit(Int) | LimitPrepare(Int) | PassedLastSwitch | Crash | ChangeResp(Zugfolge) | ...
```

```
interface Node {
    Edge getOut(Edge e);
    List<Info> triggerFront(Train train, Edge edge);
    List<Info> triggerBack(Train train, Edge edge);
}
class NodeImpl(Int x, Int y) implements Node {
  List<TrackElement> belongs = Nil;

  List<Info> triggerFront(Train train, Edge edge) {
    List<Info> ret = Nil;
    foreach (TrackElement e : belongs) {
      Info last = e.trigFront(train, edge);
      if (last != NoInfo) ret = Cons(last,ret);
    }
    return ret;
  }
  ...
}
```

Figure 2: Selected part of the implementation of the Topology Layer

The first definition represents dummy information for elements that do not transfer information to the train, but to logical elements, such as train detection devices. The following three definitions represent the information transmitted by signals, pre-signals, and the point of last visibility. The second line defines the information transmitted by a speed limiter and its announcer. `PassedLastSwitch` is transmitted at a special point marking the end of a station (the last point of danger a train passes when leaving the station), where the train may accelerate to cruising speed. Hence, it does not have a parameter for the target speed. The last line also defines two special cases: `Crash` is transmitted when a train enters a node with a bumper and `ChangeResp` at the point where the responsibility changes from one station to another. This is not a physical device on the track, but a *virtual* PIF positioned at the end of a station.

4.4. Physical Element Layer

The second layer represents track side elements. Each track element corresponds either to a physical device that allows information flow or to a *virtual element* that is responsible to model information flow at a specific distance from a physical element. For example, for each pre-signal there is a point, where it is visible from at the latest. Each element of this layer is assigned to one node of the topology layer. In case several devices are at the same position, e.g., pre- and main signal, a node at the topology layer has multiple track elements assigned.

Figure 3 shows the interface `TrackElement`, which all classes for physical elements implement. Additionally, it show the class for the contact magnet triggering the speed limiter and the speed limiter signal itself. The interface has two methods `trigFront` and `trigBack` that occur in the methods `triggerFront` and `triggerBack` of `NodeImpl`.

Every physical element in need of additional methods implements a derived interface. A `Magnet` has a reference to the logical element it belongs to (a speed limiter or self-blocking logical signal) and thus an additional setter method. A speed limiter main signal, implemented by `SpeedLimitSigImpl`, has additionally a method to set the speed limit. The `SpeedLimitSigImpl` class is an example of information flow from infrastructure to train: it transmits the information `Limit(this.allowed)`, which expresses that from now on the train must drive with at most `this.allowed` $\frac{m}{s}$. The `ContactMagnetImpl` class is an example for information from train to infrastructure: it only transmits `NoInfo` to the train, but calls the method `passed` of its logical element to send the message that it has been passed.

4.5. Logical Element Layer

The third layer groups physical elements from the second layer. For example, a pre-signal, a main signal, three ATP magnets, one visibility point for the pre-signal and at least one axle counter (at the guarded

```
interface TrackElement {
  Info trigFront(Train train, Edge edge);
  Info trigBack(Train train, Edge edge);
}

interface Magnet extends TrackElement {
  Unit setLogical(Magnetable log);
}
class ContactMagnetImpl(Magnetable logical}
      implements Magnet {
  Info trigFront(Train train, Edge e) {
    if ( this.logical != null ) this.logical!passed
  (this);
    return NoInfo;
  }
  Info trigBack(Train train, Edge e) { return
  NoInfo; }
}
```

```
interface SpeedLimitSig extends TrackElement {
  Unit setAllowed(Int i) {
}
class SpeedLimitSigImpl(Edge waitEdge)
      implements SpeedLimitSig {
 Int allowed = -1;
 Info trigBack(Train train, Edge e) {
   Info info = NoInfo;
   if (this.waitEdge == e && this.allowed >= 0 ) {
     info = Limit(this.allowed);
   }
   return info;
 }
 Unit setAllowed(Int i) { this.allowed = i; }
 Info trigFront(Train train, Edge e) { return
  NoInfo; }
}
```

Figure 3: Selected part of the implementation of the Physical Element Layer

```
class SpeedLimiterImpl(SpeedLimiter anz,
    SpeedLimiterPrepare vanz, Magnet m1, Magnet
  m2,
    Rat distance) implements SpeedLimiter {
  Int limit = -1;
  Unit setLimit(Int i) {
    limit = i;
    anz.setAllowed(i);
    vanz.setAllowed(i);
  }
```

```
  Unit passed(Magnet m) {
    if (m == m1 && limit > 0) {
      m2!activate();
      Rat time = distance/limit;
      await duration(time, time);
      m2!deactivate();
    }
  }
}
```

Figure 4: Selected part of the implementation of a speed limiter object on the logical element layer

danger point) are grouped into a single *logical* signal. The rulebooks are partially defined in terms of such logical elements. For example, setting a signal to "Go" means to set the pre-signal's *and* the main signal's *and* the magnets' state. Each physical element can be assigned to multiple logical elements, for example, a pre-signal can be assigned to two logical signals with two different main signals, or to no physical element, such as any physical element that never changes its state, like gradient changes.

Figure 4 shows the logical speed limiter that has six fields: The physical speed limiter `anz` which marks the position after which the speed limit is in place, the physical speed limiter announcer `vanz` which marks the position where the train must start braking to reach target speed at the speed limiter, a magnet `m1` which is at the same position as `anz` and activates the second magnet `m2`, the `distance` between both magnets, and the speed `limit`. The method `setLimit()` is called by a train station: depending on the line where a train leaves a station, the speed limit can differ. If the speed limiter has a fixed limit, then this method is called only once during initialization. Method `passed()` is called by the first magnet after checking that it currently displays a valid speed limit (a deactivated speed limiter has its `limit` set to -1), then commences with the activation/waiting/deactivation protocol described above.

4.6. Information Flow

Figure 5 shows the lower three layers of an entry to a train station with one entry signal and one switch. We refer to edges between two nodes as *tracks*, to the set of tracks between two signals as a *section* and to the set of tracks between the exit signal of one station and the entry signal of another as a *line*. There may be multiple lines between two stations. As described, edges do not carry any information besides their length. Gradient and other information about attribute changes are all transmitted via PIFs. E.g., instead

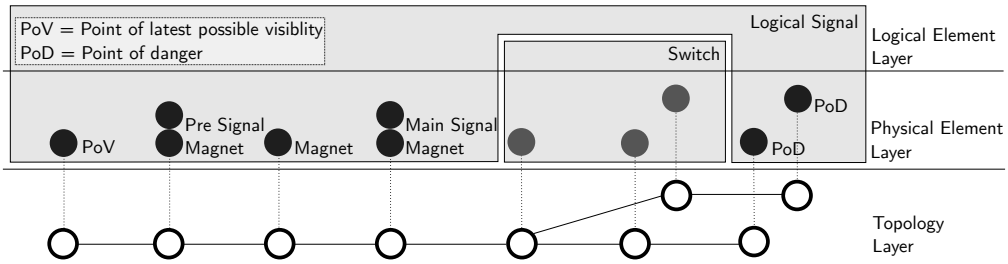


Figure 5: Illustration of the lower three layers of a logical signal in the layer model.

of modeling a tunnel as an attribute of edges, the entrances to the tunnel are two physical elements that transmit the information that a tunnel is entered and left, respectively. Each logical element resides in its own singleton (single object) COG. All nodes, edges and physical elements share one COG.

4.7. Interlocking Layer

The forth layer describes the interlocking logic and the communication between train stations.

4.7.1. 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 and do not model, e.g., intermediate signals. The rulebooks 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) which are able to "store" trains and rearrange their sequence. The generalization of both is *Zugfolgestelle* (Zfst). In the following, we use the term "station" for Zmst.

Stations manage logical elements and encapsulate the interlocking system, as well as the communication with other stations. The interlocking layer is the only spot in the infrastructure model where different elements of the same layer communicate with each other.

In general, each logical element is assigned to one train station. Each signal is assigned to exactly one Zfst managing it and every switch is assigned to exactly one Zmst (station). The Zmst is responsible to set the switches and signals correctly when a train passes. Some logical elements may not be assigned to a train station: the speed limiter described above can have a fixed speed limit and in this case requires no interface to the interlocking system.

We model the hierarchy of Zfst, Blockstelle, and Zmst using interface inheritance: Figure 6 shows the structure of the module for the interlocking layer. ABS does not have class inheritance, but every Zmst is also a Zfst and thus the implementing classes do not merely share the interface, but also the implementing methods. To avoid code duplication, ABS uses traits [12, 17]: a set of methods which can be included in a class before type checking. This does not make the analyses more complex (as inheritance would), because **super** calls are not allowed and malformed code (e.g., including a method twice) is caught by the type checker.

4.7.2. Communication

Communication with Trains. In general, a station does not communicate directly with the trains (but via infrastructure elements). It does, however, know 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 brake 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.

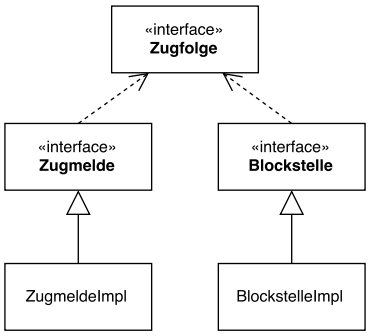


Figure 6: Class diagram for the interlocking layer

Communication along the Element Layer. As a station only communicates downwards with the logical objects, it relies on the proper setup of the lower layers. Consider an entrance signal: A station is merely notified that the train detection device covering the danger point of this signal was triggered. It relies on (1) the train detection device having been set up at the correct position and in the correct direction, (2) the train detection device having been assigned to the correct signal and (3) the line covered by the signal being the same that is listed as covered at the station. Similarly, the neighborhood of a station is encoded as a list of triples, each containing the next station, the departure signal and the connecting line. A model relies on this information being correctly provided. Checking those properties, however, is the responsibility of tools checking the consistency of the input data, e.g., RailCons [28], and their translation into an ABS model.

Communication Between Stations. The task of a Zfst is to ensure *safety of train operations*. In ETCS L1 LS the notion of safety refers to *Fahren mit festem Raumabstand*[3] and is phrased as:

There is always a signal in state "Halt" between two trains.

Safety is not achieved locally in one station. Communication between multiple Zmst is necessary to ensure that two trains do not enter the same line (which may consist of a single section, i.e. there is no further signal in between) in opposing directions. 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 the 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. The signal cannot be set to "Go" again, as long as the section it covers is locked. It must be unlocked by receiving the "*backblock*" message from the subsequent signal. That signal in turn can only send "*backblock*" after the train passed. This is one of the measures preventing a track section being occupied by more than one train.

[3]Block signaling, literally "driving in fixed intervals".

```
interface ZugFolge {
  Unit backblock(ZugFolge zf);
  Unit preblock(ZugFolge zf, Strecke s);
  Unit triggered(Signal s);
  Unit nextZfFailed(BlockStelle zf, Strecke st);
  Unit reqFree(BlockStelle zf, Strecke st);
}
interface BlockStelle extends ZugFolge {}
interface ZugMelde extends ZugFolge {
  Pair<Train, Strecke> offer(Train train, Strecke st, ZugMelde swFrom);
  Unit announce(Train train, Rat dur, ZugMelde swFrom, Strecke st);
  Unit reportback(ZugMelde swFrom, Train train, Strecke stMelde);
  Pair<Bool, Strecke> reqPermit(ZugMelde sw, Strecke stMelde);
}
```

Figure 7: Interface for Blockstelle, Zmst and Zfst (initialization methods are omitted).

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 look at the following one: If station A does not have the token for a line, A must request the token 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. To illustrate the modeling, we assume that all trains are reported back.

The description for the first two protocols is described in internal technical document, as they are implemented by the interlocking system. An overview can be found, e.g., in [8]. Procedures in case of fault in the technical systems are described in *Ril 408*. The last protocol is fully described in *Ril 408*.

Figure 7 shows the interfaces for Zfst and Zmst. The methods responsible for locking stations (`preblock` and `backblock`) are part of the `ZugFolge` interface as Blockstelle is also involved in this protocol, whereas the methods for the two other protocols (`offer`, `announce`, and `reportback` for accepting and reporting back trains; `reqPermit` for the permit token) are only relevant for Zmst.

The `ZugFolge` interface of Zfst has also a method `triggered` which relays the passing of a train at a node with a point of danger. This method completes the information flow upwards in the four layer model. The methods `nextZfFailed` and `reqFree` are used to model the error case we present in Section 5.

4.7.3. Internal State

The rulebooks do not describe the inner workings of an interlocking system. Hence, our modeling assumes as little about a Zfst as possible and aims to describe it as a black box with an executable model. We follow the technical description given in the internal training documents of DB Netz AG. Figure 8 shows the six main fields in the state of a Blockstelle:

- Field `outDesc` is a list of triples, each of which describing one exit signal of the Zfst by the (1) exit signal, (2) the line beyond the signal and (3) the Zfst at that line's end. The field `inDesc` is defined analogously for entry signals.
- The map `inLocked` maps entry signals to a boolean value which models whether they are locked. This information is not part of the logical signal: The locking is implemented physically in the interlocking

```
1 List<Triple<ZugFolge, Strecke, Signal>> inDesc  = Nil;
2 List<Triple<Signal, Strecke, ZugFolge>> outDesc = Nil;
3 Map<Strecke, Bool> inLocked  = EmptyMap;
4 Map<Strecke, Bool> outLocked = EmptyMap;
5 Map<Strecke, Bool> broken    = EmptyMap;
6 Map<Signal, Bool> brokenHere = EmptyMap;
```

Figure 8: Selected fields of a Zfst.

itself and we follow this implementation detail at this point. The map `outLocked` is defined analogously for exit signals.

- The map `broken` maps lines to boolean values which model whether the signal *at its end* is broken. This information is added because a broken signal cannot be pre-blocked and the responsible station must be contacted. We provide further details for this error case in Section 5.
- The map `brokenHere` maps a signal to a boolean value modeling whether it is broken.

A Zmst has additional fields to manage parking trains, the additional protocols and schedules. Again we assume as little internal structure as possible. Observe that since we *model* railway operations and do not reason about their *implementation* we are not forced to use efficient data structures. Rather, it is sufficient to use only simple data structures like lists and maps. The most important fields of Zmst are:

- `Map<Strecke, Bool> permit` maps lines to a boolean value modeling the permit token.
- `Map<Strecke, Bool> permitLock` mapes lines to boolean values modeling whether the permit token is locked. If it is locked, a request from another Zmst to acquire it is denied.
- `Map<Strecke, List<Train>> expectOut` maps lines to the list of trains which have been announced (and thus offered and accepted), but have not arrived yet.

A schedule consists 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, Zmst A 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.

The upper part of Figure 9 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 (represented by field `nextM` in the code) granted the request for the token. Line 6 ensures that A does not lose the permit while waiting for B to accept the train, by explicitly forbidding it (allowing it again in Line 10). Line 7 offers the train to B and line 8 notifies about the impending departure. Line 9 suspends the process until the next section is unlocked. The code in the lower part of Figure 9 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.

4.8. Trains

Trains have two positions, front and end, each modeled as the distance on a track relative to the most recently passed node. For example, if the front of a train is on track e, 5m behind a node n, then this position is described as $(e, n, 5)$. A train has a speed, an acceleration state (see Example 3.1) 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. All information flow during normal operations happens either when the front or the end of the train passes a PIF. This justifies our modeling of tracks, that keeps edges as simple as possible.

```
// ... extract correct signals and sections
if (!lookupUnsafe(permit, outSection)) {    // Zmst does not have permission
  await nextM!reqPermit(this, outSection); // acquire permit token
  permit = put(permit, outSection, True);
}
permitLock = put(permitLock, outSection, True); // lock token
await nextM!offer(train, this); // offer
nextM!announce(n, lookupUnsafe(duration, nextM), this, A); // announce
await !lookupUnsafe(outLocked, outSection); // wait until next section is free
permitLock = put(permitLock, outSection, False);
// ... set train as departed and set signal to "Go"
```

```
Unit reqPermit(Zugmelde sw, Section sec) {
  Section rt = getOtherEnd(lines, sec);
  await lookupUnsafe(expectOut, rt) == Nil && lookupUnsafe(permit, rt);
  permit = put(permit, rt, False);
}
```

Figure 9: Protocol of the offering station and for releasing the token.

Edges maintain a reference 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 the last is not lost.

4.8.1. Communication

A train receives information about the current state of a physical element in two situations: (1) It passes the node the physical element it belongs to or (2) it passed the point of earliest visibility of some physical element before and is notified about a change of the observed track element. There is no direct communication between the train and the controller. Communication between trains and logical elements occurs only in the aforementioned case that a physical element changes its state while it is observed by the train. For example, after passing the point of visibility of a pre-signal, a train is registered as an *observer* at the logical element. Now every change in the state of the logical element causes a specific information flow from the signal to the train, which is not bound to the node. After the train passes the main signal, it does not see the pre-signal anymore and is deregistered. This fits into the layered model: The logical element still serves as an interface to distribute a message from the station that controls a train to lower layers, and the train also must receive this message. The *identity* of the train is not passed to the station. The train only initiates the communication event that it passed a node, the station does not receive the identity of the train, only the information that it passed a logical element.

When a train passes a node, the method `triggerFront()` or `triggerBack()` is called (depending on which part of the train passes the node). To model the communication protocol, then either `trigFront()` or `trigBack()` 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 or trigger a train detection device.

4.8.2. Internal State

Our model of trains is based on ideas from *discrete event simulation* [29]. Each train generates a *trace*, a series of simulation events. Each such event is a triple: the event type, the position where it occurred, and an information flow that changed the inner state or behavior of the train. The simulation aims to generate the events for all trains in a distributed manner, in the correct order and correctly labeled with time.

During driving, a train computes its new state at every PIF where it is active (for example, set the acceleration state to brake). It also computes the next event and the time until this event must be processed.

Definition 4.2 (Simulation Event)**.** *A* simulation event *is an event which may cause a train to change its state or behavior. There are four 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*
- *The On-Board Unit (OBU) forces the train to brake*

The last two cases are not associated with PIFs. The last case covers the following procedure: When a train passes a pre-signal set to "Halt" then a magnet activates the OBU. The OBU monitors the speed of the train and enforces an emergency brake if the speed is above the maximal allowed braking curve or below the speed of the minimal allowed braking curve (Figure 10). The corresponding simulation event computes the intersection between the braking curve of the train and the allowed braking curves. If such a simulation event is processed, the train is forced to brake. If the train violates the lower braking limit, i.e. stops too early, it enters a restricted driving mode for a fixed distance and has a reduced maximal speed.

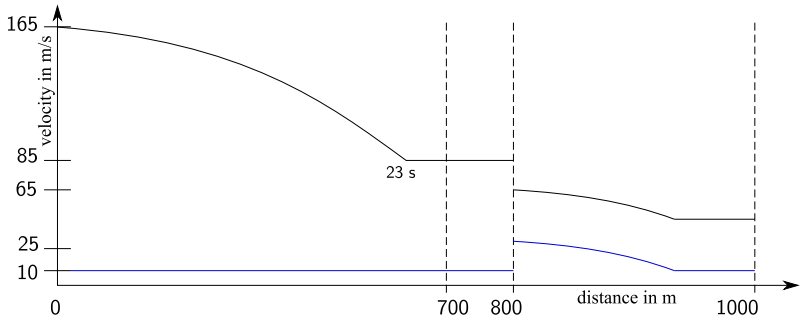


Figure 10: Allowed maximal speed during braking under the German PZB90 train control. The lower line indicates a speed limit, which starts a special restricted mode for trains when crossed. The restricted mode has a lower maximal speed.

Except schedules, trains are the only active objects, meaning they start and generate new processes without being called from the outside. These classes are also the only classes that advance time. To generate simulation events, each train has a list of `EventGenerator`s. Each such generator is responsible for one aspect of the train. Currently, two generators are implemented:

- The `PZBGenerator` implements the train protection system[4] and manages its internal state. To this end, it possesses a field of type `PZBState` that models which step of the protection sequence is active, and at which position it was activated. It may generate events that change its own internal state, e.g., to update the `PZBState` after a certain distance as pictured in Figure 10, or else the state of the whole train, by forcing it to brake if it drives too fast.
- The `PhysicsGenerator` manages physical properties such as speed and acceleration. It generates the events for the next PIF and the state changes of the physical properties, e.g., the end of an acceleration.

Generators save the state needed for their aspect, e.g., the `PhysicsGenerator` saves the current speed. The train itself has the remaining parts of its state, e.g., received and active orders. We modeled two orders so far: Order 2 is issued when the train is supposed to pass the next signal and ignore its signal aspect, Order 14.4 is issued when the train is supposed to stop at the next signal and ignore its signal aspect.

[4]Currently, for one out of three possible train modes.

The data type for modeling of events is defined as follows:

```
1 data NextEvent = Ev(Rat moment, Rat ll, AccelState newState, Rat vnew,
2          Int counter, Int fahrCount, Pos position, Time start, Rat vold, Bool pzbOneLess);
```

This definition captures information about the old state (the time `moment`, the order `counter`, the emergency counter `fahrCount`, and the old speed `vold`), as well as the `position` of the train at the change: back, front, or intermediate. When the train passes at the back or front, then the list of transmitted information items is retrieved from the node in the topology. This information is processed and results in a change of state of the train and possibly of the generators (e.g., the `PhysicsGenerator` may update its speed if the train is braking). An intermediate event carries the changes of the state: its new speed `vnew` and acceleration `newState`, its distance since the last event `ll` and a possible change of the train protection system `pzbOneLess`. Only that information is processed for intermediate events.

The `processInfo()` method processes the information coming from PIFs, see Figure 11. It is called for every information item. The information items `Mhz1000, Mhz500, Mhz2000` are issued by the magnets of the train protection system and concern the `PZBGenerator pzbGen`. Driving over an active 2000 Mhz magnet enforces an emergency brake (line 19). The `Limit` and `LimitPrepare` information items are issued by the speed limiter and speed limiter announcer, respectively. They are also relevant for the `PhysicsGenerator pGen`. Items `Info(FAHRT)` and `StartPrepare(FAHRT)` are issued by main and pre-signals, respectively. Observe that the check whether an order was issued (`listContains(orders, Ord144)`) is managed by the train itself and forces the train to stop at a main signal set at "Go" when order 14.4 was issued (line 5).

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.

Not all state changes are precomputed by simulation events. A station can issue an order at any time. In this case the train (1) computes its current state based on the current time and the most recent state, (2) changes its state according to the issued order, (3) computes, based on the state change, the next event and (4) cancels the old precomputed event.

As described in the previous section, trains drive by computing their next simulation event, then start a process that waits until the time interval associated with the event passed, before executing the state change. Orders, however, may be issued by a train station at any time and enforce a state change in the moment they arrive, for example, an order to trigger the emergency brake. This renders the old simulation event obsolete. ABS does not offer any possibility to preempt or cancel started processes. To prevent an overridden event from being executed, each process associated with a simulation event maintains a counter for the number of orders it has to process. When an event process is re-scheduled after the processed event, it first checks whether the order counter in the simulation event is identical to the one stored in the train's memory. If this is not the case, then the process terminates. The code for such methods follows this pattern:

```
1  Unit handleEvent(NextEvent ev) {
2    await duration(nextTime(ev), nextTime(ev));
3    if (counter(ev) == orderCount) { ... }
4  }
```

4.9. Discussion

Trains, tracks, stations, events, as well as the communication protocols that govern their interaction are a good match for the concurrency model of ABS and can be modeled in a straightforward and intuitive manner, as illustrated above. The concurrency model of ABS is a good match for the railway domain—components have no shared memory, and communication between multiple track elements is akin to message-passing: Communication between multiple components consist of telegrams and orders; in case of mechanical interlocking systems (ILS), pulling the wire connecting an ILS and a signal can be seen as a 1-bit message. All communication in ABS is via methods calls. This unifies the treatment of communication, as one can abstract away from the *means* of communication and only consider the communicated *information*.

Communication among different parts of one component, for example the OBU and the train driver, uses no shared memory either, but is realized via an interface of possible interactions that are mapped to

```
Unit processInfo(Information i) {
  case i {
    ...
    Info(FAHRT) => {
      if (listContains(orders, Ord144)) {
        pGen.setAccelEmergency();
        orders = without(orders, Ord144);
      }
    }
    StartPrepare(FAHRT) => {
      if (listContains(orders, Ord144)) {
        pGen.setAccelBrakeNull();
        orders = without(orders, Ord144);
        nextStopReq = True;
      }
    }
    Mhz1000 => { pzbGen.setState(Last1000, distanceTotal); }
    Mhz500 =>  { pzbGen.setState(Last500,  distanceTotal); }
    Mhz2000 => { pGen.setAccelEmergency();
                 pzbGen.setState(LastNone, distanceTotal); }
    Limit(x) =>        pGen.handleLimitEv(x);
    LimitPrepare(x) => pGen.handleLimitPrepareEv(x);
    PassedLastSwitch => { Rat vreise = pGen.getReise();
                          this.processInfo(Limit(vreise)); }
    _ => skip;
  }
}
```

Figure 11: Excerpt from the `processInfo()` method.

messages, i.e. methods. Such aggregations can be mapped to COGs: thus component parts are modeled as encapsulated objects (instead of one object per component) that do have simplified synchronization behavior, as they cannot run in parallel. Please observe that there are two possible notions of "running in parallel": Two processes are running in parallel when they are both active in a given state, but for modeling purposes we can regard two processes running on the same object in subsequent states *at the same time* also as running in parallel (if their execution order is not fixed). This is possible, because of the explicit modeling of time: event simulation time is not the same as process execution time.

In contrast to the original actor model [11], ABS uses futures to obtain the return value of a method call. This allows (1) to model the concept of "answering" directly and (2) to avoid inversion of control.

Example 4.1. *Consider the following ABS code, which is part of the model for departing a train. In line 1 the destination station is asked whether it allows the departure, in line 2 the source station waits for the answer and in line 5 the signal is set to "Go".*

```
1 Fut<Unit> f1 = nextM!offer(n, st2, this); // offer
2 await f1?;
3 nextM!announce(n, lookupUnsafe(duration, nextM), this, st2); // announce
4 List<Switch> sws = lookupUnsafe(outPaths, Pair(s, nextM));
5 this.setOutPath(sws); // set signal
```

The synchronization of `f1` *allows that setting the signal and announcing the train are performed inside one method. If the answer were to be modeled by a different message then this would separate the two actions, which are better modeled in a single method, as they are part of the same protocol.*

The concept of a PIF is central to our model and enables the event-driven structure of train driving.

PIFs are not built into ABS, but specific to the railway model. ABS offers a general and flexible modeling framework, including strong encapsulation and asynchronous communication, but the modeler still needs to find suitable domain-specific concepts. We illustrated this for two central concepts of the railway model.

Fault model: In ABS semantics faults are *failures of computation*: A process fails by raising an exception and the recovery block of its class is invoked. This allows to restore invariants that provide guarantees for other classes and processes.

Faults in railway systems are *failures of infrastructure*: A component is broken and does not work as intended. The guarantees it is supposed to provide to other components ("when the pre-signal is set to "Halt", so is the main signal") do not hold anymore. Therefore, it is inadequate to model component faults with ABS faults. Instead, we introduce a specific state to each component to model its faults.

Figure 12 shows the method `sperren()` of the logical signal. It is used to set the signal to "Halt". The field `broken` models the fault that a signal is locked at its current state. It is returned to the interlocking system to inform it that the operation failed. The field can be set in two ways: (1) in the initialization block to let the fault occur at a specific time in a specific scenario, or (2) randomly with a certain probability after each operation to model general faults.

Driving without intermediate PIF: The situation that two trains have no PIF between them does not occur in railway operations, unless a fault happens. Our model must be able to handle this case to reason about such a situation. The main problem is that, when the second train enters the edge which already contains another train, the *simulated* state shows them both as standing with their back at the last node until the first train reaches the next node. To avoid this, edges keep a list of trains that occupy them, and force the trains to update their states periodically. When a train updates its state it is handled the same as receiving a spontaneous order (i.e. the order count is increased). This does not change the state of the train, but forces the simulation to apply an intermediate simulation event.

```
1 Bool sperren(Time t) {
2   if (!broken) {
3     hs.setState(STOP, t);
4     ...
5   }
6   return broken;
7 }
```

Figure 12: Setting a logical signal to "Halt".

5. Dynamic Analysis

ABS models with initialization blocks are executable and can be compiled into Java, Haskell, Maude, ProActive, or Erlang. The ABS concurrency model described in Section 3 is implemented as a runtime environment in the target language. In this section we show that execution of ABS models is suitable to analyze the dynamic behavior of a concrete track plan. The object-oriented paradigm plus the delta and trait facility of ABS allows to vary the behavior and to perform comparisons among different model variants 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 based on the following diagram:

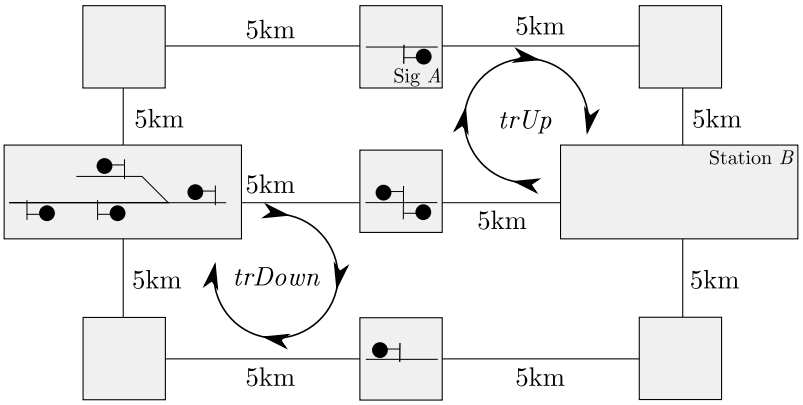
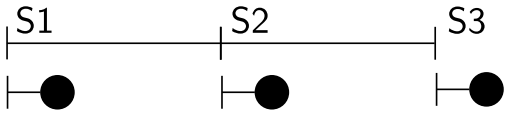


Figure 13: A Track Plan.



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

1. The controller C2 responsible for signal S2 communicates to the controller C1 responsible for signal S1 that signal S2 cannot be set to "Stop".

2. When a train arrives at Signal S1, then C1 requests a *Räumungsprüfung* (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"

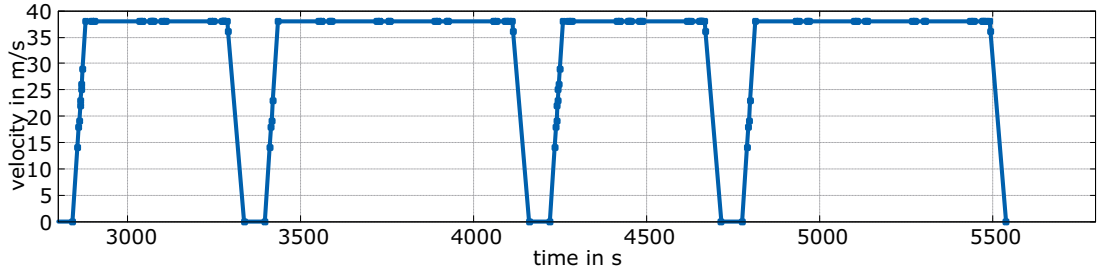
   Order 14.4: Stop at signal S2, despite S2 signaling "Go"

4. Once the train arrives at signal S2, C2 issues an *Order 2* to pass signal S2.

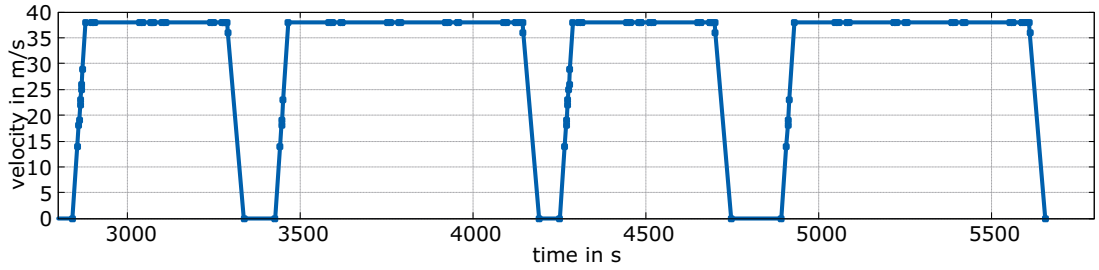
This communication protocol has four end points (including the controller responsible for S3 who ensures that the track between S2 and S3 is clear). It cannot be represented and, therefore, is not analyzable in a model that is focused on a single interlocking station. According to *Ril 408.0411*, the train must always halt before it can receive orders directly from a controller: *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 *Räumungsprüfung* only ensures that the section is clear and that the switches are set correctly, but not that the switches are fixed. Hence the train must halt to give the dispatcher an opportunity to fix the *Fahrstraße* (train route) correctly. Without this fixing, the entry signal can not be set to "Go" and the switches might be changed. 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 the whole item 4 in the communication protocol is left out.

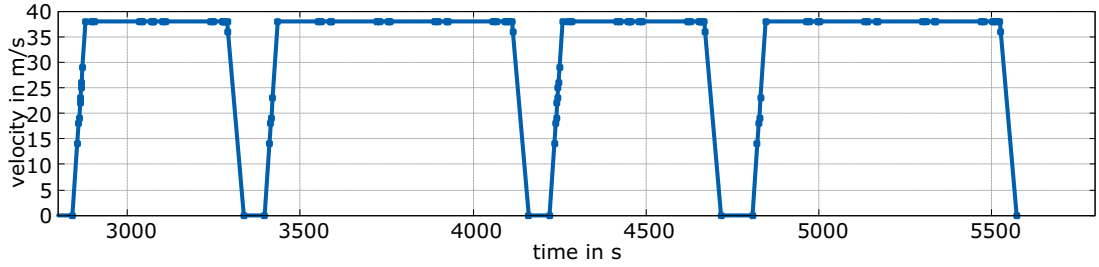
Changes in rulebooks incur considerable expenses caused by safety analysis, training, certification, etc. To decide whether this is justified, one has to estimate the expected capacity increase. Capacity is hard to



(a) Without Fault



(b) With fault, original protocol



(c) With fault, optimized protocol

Figure 14: Comparison of emerging train behavior in case of a faulty signal on a track.

determine and always requires a concrete track plan and schedule [30]. As a proof of concept for our approach, we modeled a simple track plan with two train stations and seven block stations on two intersecting circles with a length of $37km$. The track plan is shown in Figure 13. On each cycle a train is driving clockwise, thus the middle line is used in both directions. We model that the described fault occurs at signal A (causing delay for *trUp*) and show that we can analyze the delay of the train *trDown* on the *bottom* circle, given that *trDown* has to wait for *trUp* at station B. It is worth stressing that we analyze *emerging*, global behavior.

Figure 14 shows the v-t[5] diagrams for *trDown* in all three scenarios (no fault, original rule, modified rule) for the last two rounds (out of four). Without fault, every halt at station B takes 30s as scheduled. With the fault and the original protocol, the final stop at station B lasts 116s and the stop at station B before 92s. The total delay for two rounds at the final arrival in station A is 147s. With the optimized protocol the second to last stop lasts 49s and the last stop 73s, the total delay being 62s.

As discussed in Section 4 our model combines macro- and microscopic modeling. The dynamic analysis takes advantage of this. The result is expressed in *macroscopic* terms: the delays are discussed with respect to stops in a single station. The changes, however, are *microscopic*: the changed operational rules apply to single signals and the delays are caused by simulation based on a model with explicit main and pre-signals. We are able to predict macroscopic effects stemming from microscopic changes.

The ABS model of the original as well as of the optimized protocol is shown in Figure 15: in the method that models a Zfst setting its signal to "Go", we simply issue one order less to the train than before. This

[5] velocity vs. time

model transformation is not a behavioral refinement and cannot be captured in refinement-based formalisms. ABS offers *software product lines* as an effective method to manage and track model changes, see [18] for a detailed discussion. In this specific example, the fault itself is deterministic. It is part of the input which signal breaks at what point in time. The whole simulation of ca. 1.5 hours event time takes less than one second of user time 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 accelerate to maximal speed with maximal acceleration and hold maximal speed until it brakes with maximal braking power. The track plan is not realistic, for example, it is a closed network. However, it demonstrates that our modeling approach can be employed to analyze the effect of rule changes and especially the propagation of delays. In the future we intend to enrich our model with realistic speed parameters, simulating the average behavior of train drivers. As explained in Section 4.2, the track plan in our ABS model is encapsulated in a graph. It is possible to generate this graph automatically from actual infrastructure 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]);
```

Figure 15: ABS model of original and optimized protocol at Zfst with one faulty signal.

6. Static Analysis

The EN 50128 [1] standard recommends the usage of formal methods in software development for railway control systems. Our approach is a model at the *architectural level*, i.e., we abstract away from the concrete software and hardware. 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 [31]. In this section we argue that also railway systems benefit from using formal methods at a high abstraction level.

As pointed out in the introduction, safety properties in general 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 in Section 3.6.2, the strength of ABS's concurrency model is that it allows to decompose COG-global invariants into local ones which are then checked separately for each method. 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 *independently of 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 controllers: To transfer the token, the controller of station A that currently does not have the token calls the controller of station B and requests transfer. The controller of B may only release the token when all trains that departed from B have been reported back.

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 out signal.

A.2 Tracks have length strictly greater than 0.

Then we can restate property (1) as follows:

"Let S be a section between two Zmst A, B. If A releases the permit token for S, then all trains that departed A on S have been reported back." (2)

The difference between property (1) and property (2) is that property (1) describes a *state*, while property (2) describes a *history* of states, i.e. the sequence of states before the current state. Property (2) can be expressed as a history invariant which is then formalized in first-order logic and can be verified with the help of KeY-ABS [20]. In the following formula let A, B be two Zmst and S, S' two sections of a line L such hat S is the first section of L from A and 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:

$$\begin{aligned}
&\forall\, \mathsf{Int}\ i;\ \big(\exists\, \mathsf{Fut}\ f;\ h[i] \doteq \mathsf{futEv}(A, f, \mathsf{reqPermit}, (B, S))\big) \rightarrow \\
&\quad \forall\, \mathsf{Train}\ t,\ \mathsf{Int}\ j;\ j < i \rightarrow \\
&\qquad \Big(\big(\exists\, \mathsf{Fut}\ f,\ \mathsf{Rat}\ d;\ h[j] \doteq \mathsf{invREv}(A, B, f, \mathsf{announce}, (t, d, A, S))\big) \rightarrow \\
&\qquad\quad \exists\, \mathsf{Int}\ k,\ \mathsf{Fut}\ f;\ h[k] \doteq \mathsf{invREv}(B, A, f, \mathsf{reportback}, (B, t, S')) \wedge j < k < i\Big)
\end{aligned} \tag{3}$$

Theorem 6.1. *Invariant* (3) *holds for method* `reqPermit()` *in Figure 9 (and all other methods in its class), i.e. if it holds at the start of the method, then it is re-established after termination.*

This theorem is a macroscopic property, as it only reasons about lines and stations, not single edges and signals. One can see assumption A.1 as a condition on the lower three layers, i.e. a condition on the microscopic modeling which the macroscopic interlocking layer refers to.

The theorem does not yet establish that there are never two trains on one line in opposing directions. To show this one must additionally show 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 6.1 is in the Appendix. In the proof the global invariant (3) is split into two local invariants, which have been proven mechanically with the help of KeY-ABS. We also verified a more complex protocol used in certain older interlocking systems in [32]. The more complex protocol features more involved proofs, but offers no additional insights into modeling. For the sake of presentation we present the simpler protocol here.

7. Related Work

The work closest to ours is by James et al. [33], who present 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 L1 LS, which is the most wide-spread safety system in the network of Deutsche Bahn AG. Maude is an object-oriented language based on term rewriting and one of the backends supported by ABS. Potentially, both modeling approaches might be combined.

A low-level approach to model railway signaling and operations is presented by Meyer zu Hoerste [34] using Petri nets. The work provides a generic system model, concentrating on the sharing of responsibilities among humans and automated systems during railway operations.

Hoeppner [35] developes a methodology for description of railway operating processes in a generic fashion. The approach models and describes vehicle movements based on the simplest possible rail network in an UML activity diagram. As the modeled network gets gradually larger the UML activity diagrams get correspondingly larger, until a state is reached where additional extensions cause no changes in the UML activity diagrams. The methodology is used to characterize a set of basic railway operating rules (e.g., dispatch of a train from a platform).

Individual rail yard components such as interlocking systems have been analyzed by multiple approaches, for example, recently in SystemC [36], OCRA [37] and CSP||B [38]. An overview over approaches for interlocking systems, the most frequently analyzed component, can be found in the survey of Fantechi et al. [39] and a comparison of ABS with these approaches in [18].

Our approach to verification is compositional in the sense that we compose the behavior of an arbitary amount of elements to a safety guarantee for the whole system. The system is decomposed according to the *kind* of elements, such as signals and stations. Other approaches decompose a system based on its *topology* [40, 41, 42].

When it comes to the combination of micro- and macroscopic models, we are aware of two possible strategies:

1. Several models with increasing level of abstraction are related to each other. Then the most suitable one is taken for a given use case. The realization of this approach is either by generating more abstract models on demand from a microscopic model or by annotating the relation between a micro- and a macroscopic model [43, 44].

2. In mesoscopic modeling one aims at a middle ground in terms of abstraction, tailored to a use case. One recent application of this approach is timetable generation [45].

Our modeling strategy leans towards mesoscopic modeling, as it aims to combine micro- and macroscopic modeling. In contrast to other mesoscopic modeling approaches we realize this combination not by tailoring it to a single use case, but to a family of use cases that are all characterized by transmission of information. We combine micro- and macroscopic modeling in a four layer model, where the three lower layers are microscopic, the top layer is macroscopic, and the micro- and macroscopic layers reference each other.

Our approach reduces model complexity not by summarizing multiple elements, but by abstracting away from certain aspects. For example, we do model each magnet of the train protection system PZB, but we model them with zero length. On the other hand, previous mesoscopic modeling approaches do not consider the communication protocols, which is the main focus of our work, but they focus on properties relevant for civil engineering, such as the model range or precise track geometry.

Less recent examples are Abstract State Machines [46] (ASM), used by Siemens to simulate railway time tables [47] or Winter & Robinson [42] to model interlockings. Like ABS, ASMs can be compiled into mainstream programming languages and permit execution, as well as verification. ASMs follow the first combination strategy and consist of a sequence of more and more refined machines, so the verification focus is on correctness of *refinement*. Existing ASM railway models are not concerned with operations, but were written to construct railway time tables by simulation or to model single interlocking systems. Only recently [48] an adequate version of concurrent ASMs was defined, which is a prerequisite for modeling operations.

8. 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. Our dynamic analysis permits to observe emerging, global behavior that is the consequence of local faults.

```
MV;INIT;Train.TrainImpl:<0.151.0>;0;Graph.EdgeImpl:<0.133.0>
MV;REACHSTART;Train.TrainImpl:<0.151.0>;150/7;Graph.EdgeImpl:<0.134.0>;900
MV;LEAVES;Train.TrainImpl:<0.151.0>;500/21;Graph.EdgeImpl:<0.134.0>;100
MV;REACHSTART;Train.TrainImpl:<0.151.0>;200/7;Graph.EdgeImpl:<0.135.0>;200
MV;LEAVES;Train.TrainImpl:<0.151.0>;650/21;Graph.EdgeImpl:<0.135.0>;100
MV;SPEED;Train.TrainImpl:<0.151.0>;34;168;42
MV;REACHSTART;Train.TrainImpl:<0.151.0>;10343/210;Graph.EdgeImpl:<0.136.0>;482
```

Figure 16: An excerpt of a simulation trace. The syntax is explained in detailed in the downloadable model.

On Usability. Our model was developed by a team of railway engineers and software engineers. The railway engineers had no prior experience with ABS or training with object-oriented languages beyond basic C++ knowledge and the software engineers had no prior knowledge in the railway domain. Experiences during model development showed that ABS is close enough to programming languages so that software engineers can apply design patterns (for example, observers) and prototype ideas early [49]. At the same time, one can model railway operations naturally in ABS, so that domain experts can use the language to demonstrate and explain concepts by simulation and in code reviews. For example, a run of the simulation generates a trace of simulation events. An excerpt of such a trace is shown in Figure 16. We found these traces to be very useful for debugging and in communication with our railway partners. Railway engineers that were not involved in model development were quickly able to discuss details of railway operation rules in terms of ABS code.

On Restrictions. As discussed in Section 4 we approximate and discretize physics, thus we are only able to reason about safety of operations *as defined in the rulebooks*, not about safety with respect to physical properties, e.g., visual conditions and exact stopping distances during bad weather. Similarely, we do not consider safety with respect to actions that consciously and/or maliciously derive from operations, which we consider to be a security problem.

The infrastructure model is used to model information flow. It is not usable for civil engineering applications, for example, building and planning infrastructure. But neither is the infrastructure fully automatically derivable from infrastructure formats in actual use, such as RailML or PlanProML designed for planning applications: the grouping of trackside elements into logical elements is automizable in some cases (signals, switches), but it is not unique (for example, when subsidiary signals need to be considered). This not an issue of the modeling language, but of the planning rules.

On ETCS. This work discusses the German ETCS L1 LS variant, but a preliminary analysis of ETCS L2 and L3 has been conducted to identify challenges for extending the current model to these modes of operations. We identified the following challenges:

RBC The Radio Block Centre has a wider area of responsibility than the current interlocking system, but a scenario extension is not required for safety analysis and simulation of the critical parts of concurrent communication, such as RBC/RBC handovers. We assume that RBCs require no substantial changes in the layer model and correspond to our interlocking layer or, when the connected computer-based interlockings are modeled there, as a new fifth layer on top. Telegrams also appear to fit nicely into our model of transmitted information.

Information Flow without PIFs The moving blocks of ETCS L3 cannot be mapped to PIFs. As discussed in Section 4, we can handle a similar case to moving blocks in the PZB system, where the position of simulation events depends on the train speed and cannot be precomputed as PIFs. Preliminary modeling shows that the OBU of ETCS fits into our generated event framework of train driving.

Density of Topology Layer As in ETCS L3 only balises and switches are (regular) trackside elements, the graph contains less nodes. Based on the data we have so far, we assume that the simulation nevertheless scales to larger networks.

Future Work

Next 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 regarding acceleration and speed as well as to determine the precision of our approach in terms of train positions and phenomena such as roll out. On the safety side, we plan to provide a formalization of all incident scenarios described in the rulebooks [3] and to prove a suitable safety property for this model. With a validated model, we propose to use the simulation in optimization frameworks to evaluate timetables and other configurations, similar as the optimization in Section 5 is an analysis *on top of* the model, instead of *in* the model.

Furthermore, we plan to use analysis tools developed for ABS *software* models, such as complexity and deadlock analysis [25, 24], to examine the properties of the rulebook and for carrying out a capacity analysis.

Acknowledgements. The authors thank the interviewees for their time and valuable information and the anonymous reviewers for their constructive suggestions. 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] CENELEC, DIN EN 50128:2011, Railway applications – Communication, Signalling and Processing Signals, 2011.
[2] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A core language for abstract behavioral specification, in: B. K. Aichernig, F. S. de Boer, M. M. Bonsangue (Eds.), Proc. Formal Methods for Components and Objects FMCO, volume 6957 of *LNCS*, Springer, 2011, pp. 142–164.
[3] DB Netz AG, Frankfurt, Germany, Richtlinie 408, Fahrdienstvorschrift, 2017. April 2017: fahrweg.dbnetze.com/fahrweg-de/nutzungsbedingungen/regelwerke/betriebl_technisch/eiu_interne_regeln_ril_408.html.
[4] E. Kamburjan, R. Hähnle, Uniform modeling of railway operations, in: C. Artho, P. Ölveczky (Eds.), Proc. Fifth Intl. Workshop on Formal Techniques for Safety-Critical Systems (FTSCS), volume 694 of *CCIS*, Springer, 2016, pp. 55–71.
[5] E. Kamburjan, R. Hähnle, Formalisierung von betrieblichen und anderen Regelwerken - Das FormbaR Projekt, Scientific Railway Signalling Symposium, 2017. (In German).
[6] DB Netz AG, Frankfurt, Germany, Richtlinie 819, LST-Anlagen planen, 2017.
[7] Eisenbahnbundesamt (Federal Railway Authority), Eisenbahn-signalordnung, 2017. April 2017: https://www.eba.bund.de/SharedDocs/Publikationen/DE/GesetzeundRegelwerk/Bundesrecht/11_eso.html.
[8] J. Pachl, Systemtechnik des Schienenverkehrs: Bahnbetrieb planen, steuern und sichern, Springer, 2008. in German.
[9] DB Netz AG, Frankfurt, Germany, Richtlinie 482, Signalanlagen bedienen, 2017.
[10] Eisenbahnbundesamt (Federal Railway Authority), Eisenbahn-bau- und betriebsordnung, 2017. April 2017: https://www.gesetze-im-internet.de/ebo/index.html.
[11] C. Hewitt, P. Bishop, R. Steiger, A universal modular ACTOR formalism for artificial intelligence, in: N. J. Nilsson (Ed.), Proc. of the 3rd Intl. Joint Conf. on Artificial Intelligence. Standford, CA, USA, William Kaufmann, 1973, pp. 235–245.
[12] F. Damiani, R. Hähnle, E. Kamburjan, M. Lienhardt, A unified and formal programming model for deltas and traits, in: M. Huisman, J. Rubin (Eds.), Proc. 20th Intl. Conf. on Fundamental Approaches to Software Engineering (FASE), Uppsala, Sweden, volume 10202 of *LNCS*, Springer, 2017, pp. 424–441.
[13] R. Hähnle, The abstract behavioral specification language: A tutorial introduction, in: E. Giachino, R. Hähnle, F. S. de Boer, M. M. Bonsangue (Eds.), Proc. Formal Methods for Component-Based Systems FMCO, 2012, pp. 1–37.
[14] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, S. L. T. Tarifa, User-defined schedulers for real-time concurrent objects, ISSE 9 (2013) 29–43.
[15] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella, Delta-oriented programming of software product lines, in: J. Bosch, J. Lee (Eds.), Software Product Lines: Going Beyond, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 77–91.
[16] F. Damiani, M. Lienhardt, R. Muschevici, I. Schaefer, An extension of the ABS toolchain with a mechanism for type checking spls, in: N. Polikarpova, S. Schneider (Eds.), Integrated Formal Methods, 13th Intl. Conf., Turin, Italy, volume 10510 of *LNCS*, Springer, 2017, pp. 111–126.
[17] N. Schärli, S. Ducasse, O. Nierstrasz, A. P. Black, Traits: Composable units of behaviour, in: L. Cardelli (Ed.), ECOOP 2003 – Object-Oriented Programming: 17th European Conference, Proceedings, Springer, 2003, pp. 248–274.
[18] R. Hähnle, R. Muschevici, Towards incremental validation of railway systems, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation, 7th International Symposium (ISoLA), Part II, Corfu, Greece, volume 9953 of *LNCS*, Springer, 2016, pp. 433–446.

[19] C. C. Din, O. Owe, Compositional reasoning about active objects with shared futures, Formal Aspects of Computing 27 (2015) 551–572.
[20] C. C. Din, R. Bubel, R. Hähnle, KeY-ABS: A deductive verification tool for the concurrent modelling language ABS, in: A. P. Felty, A. Middeldorp (Eds.), CADE, volume 9195 of *LNCS*, Springer, 2015, pp. 517–526.
[21] P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, R. Schlatte, The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems, STTT 14 (2012) 567–588.
[22] ABS, ABS Documentation, 2018. `http://docs.abs-models.org/`.
[23] J. Lin, I. C. Yu, E. B. Johnsen, M. Lee, ABS-YARN: A formal framework for modeling hadoop YARN clusters, in: P. Stevens, A. Wasowski (Eds.), Fundamental Approaches to Software Engineering, 19th Intl. Conf. FASE, 2016, pp. 49–65.
[24] E. Giachino, C. Laneve, M. Lienhardt, A framework for deadlock detection in core ABS, Software & Systems Modeling 15 (2016) 1013–1048.
[25] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, G. Román-Díez, SACO: static analyzer for concurrent objects, in: E. Ábrahám, K. Havelund (Eds.), Proc. TACAS, volume 8413 of *LNCS*, Springer, 2014, pp. 562–567.
[26] E. Albert, M. Gómez-Zamalloa, M. Isabel, SYCO: a systematic testing tool for concurrent objects, in: A. Zaks, M. V. Hermenegildo (Eds.), Proc. 25th Intl. Conf. on Compiler Construction, CC, Barcelona, Spain, ACM, 2016, pp. 269–270.
[27] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich (Eds.), Deductive Software Verification—The KeY Book: From Theory to Practice, volume 10001 of *LNCS*, Springer, 2016.
[28] B. Luteberget, C. Johansen, M. Steffen, Rule-based consistency checking of railway infrastructure designs, in: E. Ábrahám, M. Huisman (Eds.), Integrated Formal Methods: 12th International Conference, IFM 2016, 2016, Proceedings, Springer International Publishing, 2016, pp. 491–507.
[29] J. Misra, Distributed discrete-event simulation, ACM Comput. Surv. 18 (1986) 39–65.
[30] International Union of Railways (UIC), Capacity (UIC code 406), 2004.
[31] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, M. Deardeuff, How Amazon Web Services uses formal methods, CACM 58 (2015) 66–73.
[32] E. Kamburjan, R. Hähnle, Deductive verification of railway operations, in: RSSRail, volume 10598 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 131–147.
[33] P. James, A. Lawrence, M. Roggenbach, M. Seisenberger, Towards safety analysis of ERTMS/ETCS level 2 in Real-Time Maude, in: C. Artho, P. Ölveczky (Eds.), Formal Techniques for Safety-Critical Systems FTSCS, Revised Selected Papers, volume 596 of *CCIS*, Springer, 2015, pp. 103–120.
[34] M. Meyer zu Hörste, Methodische Analyse und generische Modellierung von Eisenbahnleit- und -sicherungssystemen, volume 571 of *Fortschrittberichte VDI: Reihe 12, Verkehrstechnik, Fahrzeugtechnik. Dissertationen*, VDI Verlag, 2004.
[35] S. Höppner, Generische Beschreibung von Eisenbahnbetriebsprozessen, Ph.D. thesis, ETH Zürich, 2015.
[36] A. E. Haxthausen, J. Peleska, S. Kinder, A formal approach for the construction and verification of railway control systems, Formal Aspects of Computing 23 (2011) 191–219.
[37] C. Limbrée, Q. Cappart, C. Pecheur, S. Tonetta, Verification of railway interlocking, compositional approach with OCRA, in: T. Lecomte, R. Pinger, A. Romanovsky (Eds.), Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification, RSSRail, volume 9707 of *LNCS*, Springer, 2016, pp. 134–149.
[38] F. Møller, H. N. Nguyen, M. Roggenbach, S. Schneider, 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*, Springer, 2012, pp. 193–208.
[39] A. Fantechi, F. Flammini, S. Gnesi, Formal methods for railway control systems, STTT 16 (2014) 643–646.
[40] A. Fantechi, A. E. Haxthausen, H. D. Macedo, Compositional verification of interlocking systems for large stations, in: A. Cimatti, M. Sirjani (Eds.), Software Engineering and Formal Methods: 15th International Conference, SEFM, Trento, Italy, Springer, 2017, pp. 236–252.
[41] P. James, F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, H. Treharne, Techniques for modelling and verifying railway interlockings, International Journal on Software Tools for Technology Transfer 16 (2014) 685–711.
[42] K. Winter, N. J. Robinson, Modelling large railway interlockings and model checking small ones, in: Proc. 26th Australasian Computer Science Conf., Volume 16, ACSC, Australian Computer Society, Inc., 2003, pp. 309–316.
[43] Y. Cui, U. Martin, Multi-scale simulation in railway planning and operation, Promet - Traffic & Transportation 23 (2011) 511–517.
[44] International Union of Railways (UIC), IRS 30100 – RailTopoModel – Railway Infrastructure Topological Model, 2016.
[45] S. de Fabris, G. Longo, G. Medeossi, R. Pesenti, Automatic generation of railway timetables based on a mesoscopic infrastructure model, Journal of Rail Transport Planning & Management 4 (2014) 2–13.
[46] E. Börger, R. F. Stärk, Abstract State Machines. A Method for High-Level System Design and Analysis, Springer, 2003.
[47] E. Börger, P. Päppinghaus, J. Schmid, Report on a practical application of ASMs in software design, in: Y. Gurevich, P. W. Kutter, M. Odersky, L. Thiele (Eds.), Abstract State Machines, Theory and Applications, Intl. Workshop, ASM, Monte Verità, Switzerland, 2000, pp. 361–366.
[48] E. Börger, K. Schewe, Concurrent abstract state machines, Acta Inf. 53 (2016) 469–492.
[49] E. Kamburjan, R. Hähnle, Prototyping Formal System Models with Active Objects, Proceedings 11th Interaction and Concurrency Experience, ICE@DisCoTec 2018, Madrid, Spain, 20-21nd June 2018., 2018. (To appear in EPTCS).

Appendix

Proof of Theorem 6.1 (this sketch describes the main ideas for a formal proof, where the critical parts are performed with KeY-ABS):

> *Invariant* (3) *holds for method* `reqPermit()` *(and all other methods in its class) in Figure 9, i.e. if it holds at the start of the method, then it is re-established after termination.*

Proof. The relevant code parts are in Figure A.1. We show that once `reqPermit()` terminates, the list of trains that have departed but are not reported back is empty, but after `announce()` terminates, there is a state where the list is non-empty. Hence, the list must have been emptied in between, and the only method that removes elements is `reportBack()`. The `expectOut` field is a map from sections to lists of trains and the `getOther` function maps one section to the section on the other end of the line.

We show first that whenever `reqPermit(e,st)` terminates, then the following property holds:

```
lookup(expectOut, getOther(st)) == Nil                    (*)
```

This expresses that the list of departed but not reported trains back on the line with `st` as one end is empty. To do so, we show that the following is an invariant for the method `reqPermit()`:

$$\begin{aligned}&\exists\,\mathsf{Any}\ O, \mathsf{Fut}\ f,\ \mathsf{Expr}^*\ e,\ \mathsf{Expr}\ e';\\ &\quad\Big(\big(last(h) \doteq \mathsf{futREv}(O, \mathbf{self}, f, \mathsf{reqPermit}, e) \wedge\\ &\qquad \exists\,\mathsf{Int}\ l;\ h[l] \doteq \mathsf{invREv}(O, \mathbf{self}, f, \mathsf{reqPermit}, (e', st))\big)\\ &\qquad \rightarrow \exists\ \mathsf{Any}\ st;\ \mathbf{self}.\mathsf{expectOut}(\mathsf{getOther}(st)) \doteq \mathsf{Nil}\Big)\end{aligned}$$

where st is the parameter S passed to A, i.e. the first section of L from the A side. This invariant is local and can be proven with the KeY-ABS prover.

Now let i be the position required in (3), let $j < i$ be any position with

$$h[j] \doteq \mathsf{invREv}(A, B, f, \mathsf{announce}, (t, d, A, S))$$

for some f. By well-formedness of ABS execution histories there is a $j' < j$ with

$$h[j'] \doteq \mathsf{invEv}(A, B, f, \mathsf{announce}, (t, d, A, S))\ .$$

This event corresponds to a call on `announce()` and is can only have been generated by the method `process()`, the only method where `announce()` is called from. We can show that once `process()` terminates, then the following property holds:

```
lookup(expectOut, getOther(st)) != Nil                    (**)
```

by proving the following invariant for it:

$$\begin{aligned}&\exists\,\mathsf{Any}\ O, \mathsf{Fut}\ f,\ \mathsf{Expr}\ e;\ last(h) \doteq \mathsf{futREv}(O, \mathbf{self}, f, \mathsf{process}, e)\\ &\qquad \rightarrow \exists\,\mathsf{Any}\ st;\ \mathbf{self}.\mathsf{lookup}(\mathsf{expectOut}, \mathsf{getOther}(st)) \neq \mathsf{Nil}\end{aligned}$$

Again this invariant is local and proven with KeY-ABS. So there is a j'' with $j < j''$ that denotes the position of a state in the history where property $(**)$ holds and an i that specifies a state where property $(*)$ holds. We can assume that $j'' < i$ because a train can only trigger something in the destination station after leaving the departure station. At this point we use assumption (A.1) (the train can reach its destination station) and assumption A.2 (time needs to pass before the train can do so). Thus there must by a k with $j'' < k < i$ that is an invocation of a process that removes the element added by `announce()`. The only such method is `reportBack()`. □

```
Unit reportBack(Zugmelde swFrom, TrainI z, Section st) {
    Strecke rt = getOther(streckeMap, st);
    List<TrainI> l = lookupUnsafe(expectOut, rt);
    expectOut = put(expectOut, st, without(l, z));
}

// called upon arrival of n, after the route to its halting signal has been set
Fut<Unit> process(TrainI n) {
    // wait for the train to arrive and stop for haltTime seconds
    await getSigFor(parking, n) != null;
    Time last = await n!acqStop();;
    Rat r = timeValue(now()) - timeValue(last);
    await duration(haltTime-r,haltTime-r);
    Signal s = getSigFor(parking, n);

    // extract its target
    Strecke st2 = extractNextStrecke(outDesc, s);
    ZugFolge next = extractNextFolge(outDesc, s);
    ZugMelde nextM = getFromSignal(outMelde, s);

    // announce, etc.
    List<Switch> sws = await this!setPreconditions(s, n, st2, next, nextM);

    // save as departed
    expectOut = put(expectOut, st2, Cons(n, lookupUnsafe(expectOut,st2)));
    Fut<Unit> f8 = this!setOutPath(sws);
    return f8;
}

Unit reqPermit(Zugmelde sw, Section sec) {
  Section rt = getOtherEnd(lines, sec);
  await lookupUnsafe(expectOut, rt) == Nil && lookupUnsafe(permit, rt);
  permit = put(permit, rt, False);
}
```

Figure A.1: Slightly prettified, relevant part of `ZugmeldeImpl` for the proof. The full implementation is available in the downloadable model.