Formal development and verification of a distributed railway control system

Haxthausen, Anne Elisabeth; Peleska, J.

Published in:
IEEE Transaction on Software Engineering

Link to article, DOI:
10.1109/32.879808

Publication date:
2000

Document Version
Publisher's PDF, also known as Version of record

Formal Development and Verification of a Distributed Railway Control System

Anne E. Haxthausen and Jan Peleska

Abstract—In this article, we introduce the concept for a distributed railway control system and present the specification and verification of the main algorithm used for safe distributed control. Our design and verification approach is based on the RAISE method, starting with highly abstract algebraic specifications which are transformed into directly implementable distributed control processes by applying a series of refinement and verification steps. Concrete safety requirements are derived from an abstract version that can be easily validated with respect to soundness and completeness. Complexity is further reduced by separating the system model into a domain model and a controller model. The domain model describes the physical system in absence of control and the controller model introduces the safety-related control mechanisms as a separate entity monitoring observables of the physical system to decide whether it is safe for a train to move or for a point to be switched.

Index Terms—Safety, railways, distributed control system, formal specification, verification, stepwise refinement, RAISE.

1 Introduction

The present modernization of European railway networks raises a large variety of issues related to the design and verification of railway control systems. One of these problems is the question, how to design control systems for small local networks that can only operate effectively if the costs for initial installation, operation, and maintenance of the control system are low? Today's centralized interlocking systems—at least those which are available in Germany—are far too expensive for such small (possibly private) networks. A promising approach is to *distribute* the tasks of train control, train protection, and interlocking over a network of cooperating components using the standard communication facilities offered by mobile telephone providers. On the other hand, a distributed control concept also introduces new safety issues that could be disregarded, as long as centralized control was applied. First, the new communication medium requires security and reliability mechanisms that were unnecessary for centralized systems transmitting control commands to signals and points over wires. Second, the distribution of a control algorithm over several components raises new design and verification issues, since the concept of a global state space as available in a centralized interlocking system can no longer be implemented.

In this article, we will describe the concept of a distributed railway control system consisting of *switch boxes (SB)*, each one locally controlling a point, and *train control computers (TCC)* residing in the train engines and collecting the local state information from switch boxes along the track to derive the decision whether the train may enter the next track segment. The system concept does not require signals along the track, since the "go/no-go" decisions are performed and indicated in the train control computers. We give an overview of the formal specification and verification of the main control algorithm executed by the distributed cooperating control components. The system is designed to operate on *simple networks*, which means in our context that there are two distinguished destinations A and B, such that at each track segment of the network there is a uniquely defined direction to reach A and B, respectively. Typically, this definition applies to networks which are not highly frequented by trains and connect two main stations with small intermediate stations (Fig. 1).

Our specification and verification approach is based on the RAISE formal method and tool set [10], [11], and follows the *invent-and-verify paradigm*. To address safety issues in a systematic way the standard procedure (see [12]) separating the *equipment under control*—that is, the railway network with its trains—from the *control system*—in our case, the set of TCCs and SBs—is applied. To this end, we first develop abstract algebraic specifications for the *domain model*, i.e., the railway network and the trains to be controlled, and the *safety requirements* stating that the system must not perform a transition into a hazardous state where trains may collide or derailing might occur. These requirements are expressed as conditions about the *observables* of the domain model. Using stepwise refinement and accompanying verification steps, we introduce additional observables that may be monitored by a *controller* giving the "can move/cannot move" conditions for each train and the "can be switched/cannot be switched" conditions for each point. The completeness and consistency of these conditions is verified by proving refinement relations to the higher-level specifications which already have been proved to be consistent with the initial safety requirements. The first stage of the invent-and-verify development ends when the observables

- *A. Haxthausen is with the Deptartment of Information Technology, Technical Univiversity of Denmark, Denmark-2800 Lyngby. E-mail: ah@it.dtu.dk.*
- *J. Peleska is with FB-3 Informatik, Universität Bremen, Postfach 33 04 40, D-28334 Bremen, Germany. E-mail: jp@tzi.de.*

Manuscript received 14 Feb. 2000; accepted 28 Apr. 2000.
Recommended for acceptance by J. Wing and J. Woodcock.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112020.

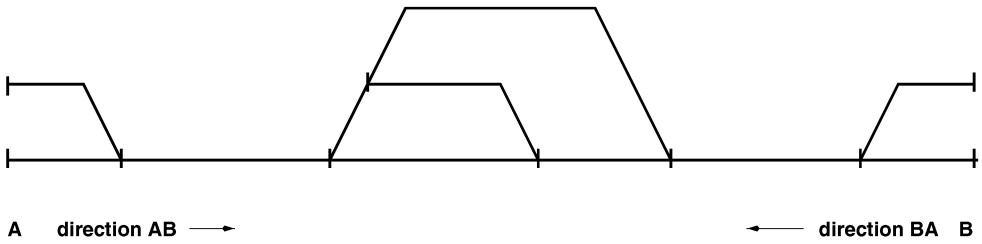

Fig. 1. Simple railway network.

of the last refinement needed to control the safety of train movements and point switching are implementable in the sense that they can be transformed into a concrete state space that may be conveniently partitioned among a set of distributed cooperating processes. The second stage specifies and verifies the concrete—i.e., implementable—distributed *controller model* by introducing communicating processes which represent train control computers and switch boxes. The TCC processes collect state information from the SB processes to make the "can move/cannot move" decisions. The SB processes store the relevant state information to take the "can be switched/cannot be switched" decisions for their local points. The resulting controller is a distributed program which is under-specified with respect to application-dependent control decisions—like defining the order in which trains may pass along a single-track section—which can be made without violating the safety requirements. Concrete controller implementations will resolve this under-specification by choosing a specific solution for application-dependent control decisions.

The work presented here originated from a collaboration of the authors with INSY GmbH Berlin, who developed the distributed systems design described in the next section for their railway control system RELIS 2000 designed for local railway networks. In this collaboration, the authors focus on the generalisation and verification of the control concepts used in RELIS 2000. Furthermore, the second author is cooperating with Siemens and Transnet (South African Railways) in the field of verification, validation and test of safety-critical systems.

In Section 2, we introduce the general concept for the distributed railway control system discussed in this article. Similar approaches of "Funkbasierter Fahrbetrieb (FFB)"—that is, train control based on radio transmission—are presently being investigated by German Railways [3]. Our verification concept described in the following sections applies to all of these approaches. Section 3 presents the formal specification of the system's domain model. In Section 4, an abstract version of the safety requirements is introduced. The subsequent sections are concerned with the development of the control system as a series of refinement and verification steps. In the discussion (Section 7), we sketch the more general issues of our concept for the development, verification, validation and test of safety-critical systems. In the Appendix, one of the safety proofs is given.

2 Engineering Concept

In this section, we introduce the technical concept of the distributed railway control system to be formally specified and verified below. The technical concept is based on the RELIS 2000 system of INSY GmbH with generalizations and modifications performed by the authors.

Consider the system configuration depicted in Fig. 2. The tasks of train control, train protection, and interlocking are distributed on train control computers residing in each train T1, T2 and switch boxes SB1, SB2, SB3, each one controlling a single point, the boundary between two segments (e.g., blocks) of a single track or a railway crossing. The basic principle of the control algorithm is as follows:

- Each switch box stores the local safety-related information in its state space. For example, this information contains the actual state of the traffic lights guarding the railway crossing, i.e., the track segments that are presently connected by the local

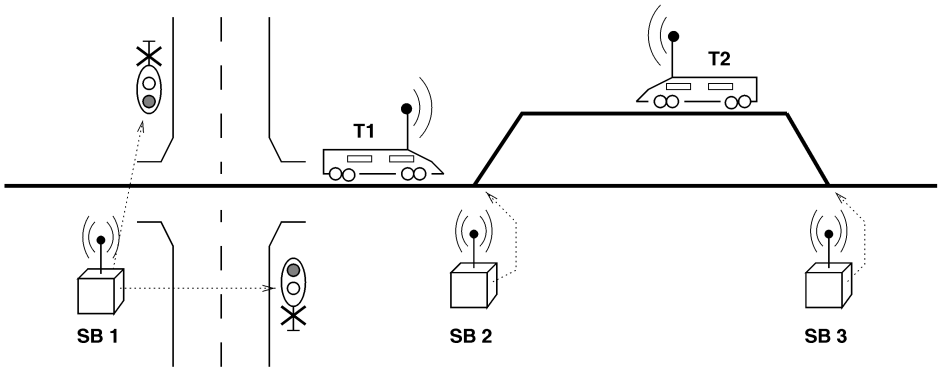

Fig. 2. Distributed railway control system—trains communicating with switch boxes.

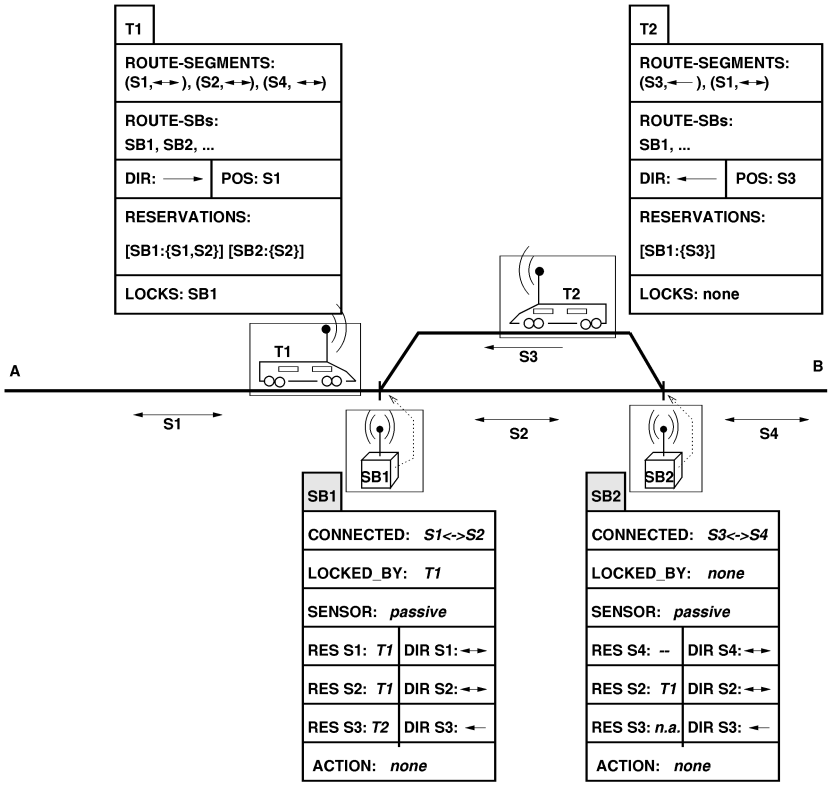

Fig. 3. Switch boxes, trains, and their state spaces.

point, or whether a train is approaching the switch box. The switch boxes use sensors to detect approaching trains and to decide whether a train has left the critical area close to a point or a crossing.

- To pass a railway crossing or to enter a new track segment, a train's TCC communicates with the relevant switch boxes to make a request for blocking a crossing, switching a point, or just reserving the relevant track segments at the SB for the train to pass. The decision which switch boxes address is based on the location of the train which is determined by means of the Global Positioning System (GPS) or by using track components signaling their location to the passing train.
- Depending on their local state, the switch boxes may or may not comply with the request received from a TCC. In any case, each SB returns its (possibly updated) local state information to the requesting TCC. After having collected the response from each relevant SB, the TCC evaluates the SB states to decide whether it is safe to approach the crossing or to enter the next track segment.
- For train protection, each TCC blocks the train engine if it is not allowed to leave a station and triggers the emergency brake if the train approaches a railway crossing or enters a new track segment without permission from the associated switch boxes. Furthermore, each TCC monitors the speed of the train and gives warning messages or triggers the emergency brakes if the actual speed exceeds the maximum velocity admitted for the type of train at its actual location in the network.

Observe that in principle, the concept sketched above would admit completely automatic train control without train engine drivers being present. However, in the possible realisations presently discussed, this is not intended. The train engine driver has the ultimate responsibility to decide whether it is safe to leave a station, enter a new track segment or pass a crossing.

In the subsequent sections, we will focus on the formal specification and verification of the control algorithm concerned with "can move/cannot move" decisions for trains and "can be switched/cannot be switched" decisions for points. To introduce the principles of this algorithm, consider Fig. 3, which shows the local state spaces of two switch boxes SB1, SB2 and trains T1, T2.

In state component CONNECTED, the switch box stores which track segments are presently connected by the local point. (If the SB just separates two blocks on a single track, this information is static.) In the components DIR S1, DIR S2,... the directions associated with each track segment are stored: A segment can either be used only for trains going in direction $A \rightarrow B$, or for trains going in direction $B \rightarrow A$ or in both directions ($A \leftrightarrow B$). Typically, this information is fairly static and will only be changed if deviations from the ordinary train schedule occur, for example when constructions are going on or when a train arrives late. As explained below, the segment direction will be evaluated to decide whether a train may reserve a switch box. The LOCKED_BY state component indicates whether a specific train has the right to pass the switch box. If such a train is registered in this component, it is impossible to switch the local point to another direction until the train has passed. For the detection of passing trains, a state component SENSOR is

activated by a set of sensors attached to the track when a train approaches the point. The component is returned to state "passive" as soon as the sensors indicate that the last wagon of the train has passed the point. To decide whether a train may get a reservation for a segment approaching the switch box and whether a point may be locked for a train, additional state components RES S1, RES S2,... are maintained at each switch box for every track segment whose segment direction is approaching the SB. The ACTION component of the state space is used as a "transaction flag" for commands which have to be executed on several switch boxes in a synchronized manner: The switch box will refuse new commands, as long as the ACTION flag indicates such a transaction. Observe that this flag is unnecessary for the standard reservation commands described next.

The state space of each TCC contains the lists ROUTE-SEGMENTS and ROUTE-SBs of track segments and switch boxes along the train route. When leaving a segment and passing a switch box, these entries are removed from the head of each list. Again, segments are stored together with their directions $\rightarrow, \leftarrow, \leftrightarrow$. State component DIR stores which direction the train is heading. A train may only move along segments whose direction is compatible with DIR. In POS, the actual position is stored. In the abstraction presented here, positions are specified by one or two segments, the former indicating that the train is on the segment without touching neighboring segments, the latter indicating that the train is in the critical area of a point (potentially) connecting the two segments. State component RESERVATIONS stores the switch boxes and associated segments which have been reserved by the train. LOCKS is a list of switch boxes whose points have been switched in the direction of the train route and are locked for the train. Whenever a train is allowed to proceed into the next segment, this information must be consistent with the corresponding RES- and LOCKED_BY-components of the switch boxes involved.

To determine, whether a train T1 may enter a new segment S2 (Fig. 3), the train control computer and the relevant switch boxes evaluate the state space described above as follows:

- To guarantee safety for the train at its local position, two conditions must be fulfilled:
  1. The train direction must be consistent with the direction associated with the local track segment. (Train T1 going in direction $A \rightarrow B$ cannot have its position on segment S3, since the latter has associated direction $B \rightarrow A$.)
  2. Each train must have a reservation for its local track segment at the next switch box to be approached by the train (S1 must be reserved for train T1 at switch box SB1).
- To enter the next segment (S2 for train T1), three safety conditions must be fulfilled:
  1. The train direction must be consistent with the direction of the segment to be entered. (S1 has direction $A \leftrightarrow B$, so this is consistent with T1's train direction $A \rightarrow B$.)
  2. The next SB must be locked for the train (SB1 is locked by T1, so this condition is fulfilled for T1). Note this can only be the case if the segment ends at the SB have been switched in the direction of the train route.
  3. The train must have a reservation for the next segment S2 at every switch box where S2 is an approaching segment. (In Fig. 3, S2 approaches both SB1 and SB2, so T1 must reserve S2 at both switch boxes. In contrast to that, T2 only needed to reserve S3 at SB1 before entering S3 from S4.)
- In order to fulfill these three conditions, the train signals its wish to enter the next segment to the associated switch boxes. Each switch box enters the train's reservation for the next segment if this is not already reserved for another train. If reservation is possible and the SB is not locked by another train, it will switch its point into the required direction if necessary and lock the point for the requesting train.
- If the three conditions are fulfilled the train may enter the next segment. As soon as the train has passed the next SB, the SB will delete the lock and all reservations made by the train. (In Fig. 3, SB1 will unlock its point and delete all references to T1, as soon as the train has passed the point and entered S2. Note that T1 is still completely safe at its new location, since each train wishing to enter S2 from either S1 or S4 also needs a reservation of S2 at SB2, and this is still blocked by T1.) The train will update its own state space accordingly.

In the sections below, this informal system concept is described and verified in a formal way. Observe that in this article we deal with untimed control and safety mechanisms only. Time-dependent conditions—for example, when the last point in time (depending on speed and position) to trigger the emergency brakes in order to prevent the train from entering the next segment—are imported into the specifications at a later stage as a "timed refinement" of the untimed control mechanisms discussed here.

3 Domain Model

In this section, we show (parts of) a domain model capturing those physical objects and events of the uncontrolled railway system which are relevant for the development of the railway control system. We divide the model into a static part and a dynamic (state based) part. Other authors have established similar railway domain models [1], [5], [6].

3.1 Static Part of the Model

The static part of the model comprises definitions of data types for physical objects. The objects we consider include the trains, the points (switch boxes), and the railway network.

Trains. Each train has a unique identification belonging to the following, not further specified type

type TrainId

Hence, our model is independent of the number of trains.

Points. Each point has a unique identification belonging to the following, not further specified type.

type PointId

Railway Network. A railway network consists of segments connected according to the network topology.Each segment has a unique identification belonging to the following, not further specified type:

type Segment

In our model, the network topology is specified by a predicate (*are_neighbors*) which defines which segment ends are neighbors.

value
are_neighbours : SegmentEnd × SegmentEnd → **Bool**

where a segment end is a pair consisting of a segment identification and one of two possible ends.

type
SegmentEnd = Segment × End, End ==
a_end | b_end

The *are_neighbors* predicate must satisfy a number of axioms (not presented here) ensuring that the network is directed.

Hence, our model is independent of the size and exact layout of the network.

3.2 Dynamic Part of the Model

As trains move along the segments of the network and points are switched, the *state* of the railway may change over time. We use a discrete, event-based model to describe state transitions.

The State Space. At this early phase of development, we do not yet know, what the exact state space is, but only that the state space should contain information about some dynamic properties of objects which we will explain below. Therefore, we just introduce a name for the type of states without giving any datatype representation.

type State

and characterize this type implicitly by specifying state observer functions of the form *obs : State* × ... → *T* which can be used to capture information (of type *T*) about the state.

Dynamic Properties of Trains. Each train has a *position* and a *direction* which may change over time.

We assume that the length of segments is chosen such that any train has a position on one or two neighboring segments[1] or it has passed an end point of the network.

1. Our engineering concept can be adapted to railway systems for which this assumption does not hold by using *lists of segments* to represent train positions. The type definition of "Position" used in this article corresponds to a representation with lists of maximum length 2.

type
Position = =
single(seg_of : Segment) | double(fst : Segment, snd : Segment) | error

A position of the form $single(s)$ indicates that the train is residing on a single segment s. A position of the form $double(s1, s2)$, where $s1$ and $s2$ are two neighboring segments, indicates that the train is residing on one or both segments in the critical area of the point potentially connecting these segments. The $error$ position is used to model the case where a train has passed an end point of the network.

Since the railway network is directed according to our *simple network* assumption described in the introduction, there are two possible train directions.

type Direction == dirAB | dirBA

We introduce the following to observe the mentioned properties:

value /∗ state observers ∗/
position : State × TrainId → Position,
direction : State × TrainId → Direction

We only consider states satisfying the *physical law* that the position pos and direction dir of any train must conform in the sense that if the position is of the form $double(s1, s2)$ then the "to-end" in direction dir of segment $s1$ must be a neighbor to the "from-end" in direction dir of segment $s2$ (in other words: the train must drive in the direction from $s1$ to $s2$). This is expressed by the following axiom:

axiom [position_direction_consistent]
∀ σ : State, t : TrainId, s1, s2 : Segment :●
position(σ, t) = double(s1, s2) ⇒
are_neighbors(to_end(s1, direction(σ, t)),
from_end(s2, direction(σ, t)))

The "to-end" in direction dir of segment s is defined as follows:

value
to_end : Segment × Direction → SegmentEnd
to_end(s, dir) ≡ **if** dir = dirAB **then** (s, b_end) **else** (s, a_end) **end**

The "from-end" is the opposite end of the "to-end."

Dynamic Properties of Points. Points may be switched. Hence, the connections between segment ends of the railway network may change over time. We introduce the following function to observe this:

value /∗ state observer ∗/
are_connected : State × SegmentEnd × SegmentEnd →**Bool**

The *are_connected* observer must satisfy some axioms (not presented here) ensuring that some *physical laws* are satisfied, e.g., that only neighboring segments are

connected and there is exactly one connection in each point.

Events. We consider the following events:

- trains move from one position to their next position, and
- points are switched.

It should be noted that in this uncontrolled model, events may lead to unsafe states.

For each kind of event, we introduce a state constructor which can be used to make the associated state changes.

```
value /* state constructors */
  move : State × TrainId → State,
  switch : State × PointId × SegmentEnd → State
```

Their behavior is defined by observer axioms. For instance, the following axiom states that moving a train does not change how segment ends are connected.

```
axiom /* observer axioms */
  [are_connected_move]
    ∀ σ : State, t : TrainId, se1, se2 : SegmentEnd •
      are_connected(move(σ, t), se1, se2) ≡
      are_connected(σ, se1, se2)
```

and the following axiom states that moving a train affects the position of the train itself:

```
  [position_move]
    ∀ σ : State, t1, t2 : TrainId •
      position(move(σ, t1), t2) ≡
        if t2 = t1 then
          next_position(σ, position(σ, t2),
          direction(σ, t2))
        else position(σ, t2) end
      pre safe(σ)
```

where $safe$ is a function defined in next section, and *next_position* is an auxiliary function defined below. The term next_position(σ, position(σ, t2), direction(σ, t2)) gives the next position of the train in its driving direction.

```
value
  next_position : State × Position × Direction →
  Position

axiom
  [next_position_of_double]
    ∀ σ: State, s1, s2 : Segment, dir : Direction •
      next_position(σ, double(s1, s2), dir) ≡
      single(s2),

  [next_position_of_single_at_end]
    ∀ σ : State, s1 : Segment, dir : Direction •
      (∀ s2 : Segment •
        ~ are_neighbors(to_end(s1, dir),
        from_end(s2, dir))) ⇒
      (next_position(σ, single(s1), dir) ≡ error),

  [next_position_of_single_at_connection]
    ∀ σ : State, s1, s2 : Segment, dir : Direction •
      are_connectedσ, to_end(s1, dir),
      from_end(s2, dir)) ⇒ (next_position(σ,
      single(s1), dir) ≡ double(s1, s2)),

  [next_position_of_single_at_non_connection]
    ∀ σ : State, s1, s2 : Segment, dir : Direction •
      are_neighbors(to_end(s1, dir), from_end(s2,
      dir)) ∧ (∀ s : Segment •
        ~ are_connected(σ, to_end(s1, dir),
        from_end(s, dir))) ⇒
      (next_position(σ, single(s1), dir)
      ≡ double(s1, s2))
```

The first axiom states that the next possible position of a train having a position on two segments, $s1$ and $s2$, is its front segment $s2$. The second, third, and fourth axiom define the next possible position for trains in direction dir having a position on a single segment $s1$. If the "to-end" in direction dir of segment $s1$ has no neighbors, the train is at an end point of the railway network and will have $error$ (modeling derailing) as its next possible position. If the "to-end" in direction dir of segment $s1$ is connected to the "from-end" in direction dir of a (neighbor) segment $s2$ then the train will have its next possible position on $s1$ and $s2$. The same holds if the "to-end" in direction dir of segment $s1$ has no connections, but has the "from-end" in direction dir of a segment $s2$ as neighbor.

There are similar observer axioms for switch.

4 Safety Requirements

Our goal is to develop a train control and interlocking system, satisfying the following two safety requirements:

No collision. Two trains must not reside on the same segment.

No derailing. Trains must not derail (by passing an end point of the network or by entering a point from a segment which is not connected with the next segment).

The notion of safety can be formalized by defining a predicate which can be used to test whether a state is safe.

```
value
  safe : State → Bool
  safe(σ) is no_collision(σ) ∧ no_derailing(σ),

  no_collision : State → Bool
  no_collision(σ) ≡
    (∀ t1, t2 : TrainId • t1 ≠ t2 ⇒
      segments(position(σ, t1)) ∩ segments(position
      (σ, t2)) = {}),

  no_derailing : State → Bool
  no_derailing(σ) ≡
    (∀ t : TrainId •
      position(σ, t) ≠ error ∧ (∀ s1, s2 : Segment •
      position(σ, t) = double(s1, s2) ⇒ are_connected
```

(σ, to_end(s1, direction(σ, t)), from_end
(σ, s2, direction(σ, t)))
)
)

Here, *segments* is an auxiliary function giving the segments of a position.

Observe that the no-derailing safety requirement above only covers wrong point positions as the cause for derailing, but does not refer to derailing due to excessive speed of trains. However, this cause for derailing can be handled by a completely separate safety-mechanism. To avoid derailing due to excessive speed, the maximum velocity is calculated as a function of the train type, the number of wagons attached to the train engine, and the actual train position. It is continuously checked whether the actual speed does not exceed the calculated maximum value, otherwise the train control computer issues a warning and may even automatically trigger the brakes. Obviously, this safety mechanism can be designed and implemented completely independent from the safety mechanisms preventing collisions and derailing due to wrong point positions. Therefore, we do not consider derailing due to excessive speed in the following sections.

The reader may have noted that our top-level safety requirements neither refer to *shunts*[2] nor to *flank protection*[3] of trains. In the discussion (Section 7), we will explain that these should be regarded as lower-level safety requirements which are needed to implement the no-collision and no-derailing requirements above, because trains cannot stop immediately.

5 Development of the Railway Control System: First Stage

The purpose of the railway control system is to prevent events from happening when they may lead to an unsafe state. We develop an implementable controller model by *stepwise refinement*[4] following the *invent-and-verify paradigm.*[5] The development is divided into two major stages of which we describe the first in this section.

In the first major stage of development, we design a full state space, keeping information not only about the dynamic properties described in the domain model, but also about new dynamic data (observables) like segment reservations which may be monitored by the controller to evaluate the "can move/cannot move" and "can be switched/cannot be switched" conditions. New data, like segment reservations, also gives rise to new state constructors modeling events like making a reservation.

Our strategy for fulfilling the safety requirements is to invent,

1. a *state invariant consistent* σ, and
2. for each constructor *con* (σ...,) a *guard* (condition) *can_con*(σ, ...) which can be used by the controller to decide whether it should allow events (corresponding to application of that constructor) to happen,

such that the following *strong safety requirements* are fulfilled:

1. States satisfying the state invariant must also be safe.
2. Any state transition made by a state constructor must preserve the state invariant when the associated guard is true.
3. If the guards for two possibly concurrent events are both true in a state satisfying the state invariant, then a state change made by one of the events must not make the guard for the other event false.

These requirements ensure that if the initial state satisfies the state invariant, and the railway control system only allows events to happen when the corresponding guards are true then the system will stay safe.

The first strong safety requirement can be formalized by the following theory:

[consistent_is_safe]
$\forall \sigma$: State • consistent(σ) $\Rightarrow$ safe(σ)

The second strong safety requirement can be formalized by a theory.

[can_con]
$\forall$ ... • consistent(σ) $\wedge$ can_con(σ, ...) $\Rightarrow$ consistent (con(σ, ...))

for each constructor *con*. The third strong safety requirement can be formalized by a theory typically of the form:

[can_con1_con2]
$\forall$ ... •
consistent(σ) $\wedge$ can_con1(σ, x) $\wedge$ can_con2(σ, y)
$\Rightarrow$ can_con2(con1(σ, x), y)

for each pair of constructors, *con1* and *con2*, belonging to two possibly concurrent events.

The state space, state invariant, guards, etc., are found by stepwise refinement and verification.

5.1 First Specification

The first specification is an abstract, algebraic specification extending the domain model with the following declarations:

value /* state invariant */
consistent : State $\rightarrow$ **Bool**
value /* guards for constructors */
can_move : State $\times$ TrainId $\rightarrow$ **Bool**,
can_switch : State $\times$ PointId $\times$ SegmentEnd $\rightarrow$ **Bool**

As the *State* is not yet explicit, and the set of observers is not complete, we cannot yet give complete explicit definitions

2. A shunt is a track section reserved in front of a train in order to prevent collisions in situations where the braking process does not succeed to stop the train at the intended position (e.g. in front of a signal switched to red).

3. Flank protection requires that points in the vicinity of a train t_1 may not be switched in such a way that another train t_2 which failed to stop at the required position on a neighboring track segment might enter the track segment reserved for t_1, thereby colliding with the "flank" of t_1.

4. That is, the model is developed in a number of steps, where each step starts with a specification and produces a new specification which is more detailed (as more design decisions have been taken).

5. That is, in each step, first the new specification is invented and then it is verified that it refines (is a correct development of) the previous specification.

of the state invariant and guards. Instead, we specify requirements to the guards by implications of the form.

axiom /∗requirements to guard can_con ∗/
[can_con_implication1]
∀ ... can_con(σ, ...) ∧ consistent(σ) ⇒ ...

and requirements to the state invariant by an implication of the form.

axiom /∗ requirements to consistent ∗/
[consistent_implication1]
∀ σ : State • consistent(σ) ⇒ p1(σ)

We use implications so that we can enrich the requirements in later steps with additional constraints.

p1(σ) is chosen to be *safe(σ)*, and the requirements to *can_move* is,

```
axiom /∗ requirements to guard can_move ∗/
  [can_move_implication1]
    ∀ σ : State, t : TrainId •
      consistent(σ) ∧ can_move(σ, t) ⇒
        case position(σ, t) of
          single(s1) →
            let dir = direction(σ, t) in
              (∃ s2 : Segment •
                are_connected(σ, to_end(s1, dir),
                from_end(s2, dir)) ∧ (∀ t2 :
                TrainId
                • t ≠ t2 ⇒
                  s2 ∉ segments(position(σ, t2)) ∧
                  ((consistent(σ) can_move(σ, t2))
                    ⇒ s2 ∉ segments(position
                     (move(σ, t2), t2)))
                )
              )
            end,
          double(s1, s2) → true,
          _ → false
        end
```

The above axiom states that if a train t is allowed to move from a single segment $s1$ in a consistent state then there must exist a next segment $s2$ in the driving direction of the train which is properly connected to $s1$ (such that moving the train does not lead to a derailing.) All other trains $t2$ are neither allowed to reside on $s2$ nor allowed to move to $s2$ (such that collisions are avoided.) The axiom also states that if a train has passed the end of the network then the state is inconsistent and/or the train is not allowed to move.

The requirements to *can_switch* are similar.

Sketched in the Appendix is a proof that this specification satisfies the following theory.

```
[safe_can_move]
∀ σ : State, t : TrainId •
  consistent(σ) ∧ can_move(σ, t) ⇒ safe(move(σ, t))
```

i.e., the axioms *[can_move_implication1]* and *[consistent_implication1]* are consistent with the second strong safety theory *[can_move]*.

5.2 Second to Fourth Specification

Each of the next three specifications are algebraic and obtained from the previous specification by adding declarations of new observers, state constructors and guards, observer axioms for new observers, and new constructors and requirement axioms (in form of implications) for new guards. Furthermore, the requirements to the state invariant is enriched in specification number i by adding the axiom,

axiom /∗ requirements to consistent ∗/
[consistent_implicationi]
∀ σ : State • consistent(σ ⇒ (pi(σ))

(where $pi(\sigma)$ is a predicate), and the requirements to some of the previous guards *can_con* are refined by making the predicate of the right-hand side of the *[can_con_implication]* axioms stronger.

Below, we give a short survey of which concepts are added in the second to fourth specification.

Second Specification. In the second specification, two new concepts are introduced.

- segment registrations for trains, and
- segment directions.

This gives rise to two new observers and types for segment registrations and directions,

```
value /∗ state observers ∗/
  registered : State × Segment → Register,
  seg_dir : State × Segment → SegDir
type
  Register = = no | train(tr : TrainId),
  SegDir = Direction | BothDir,
  BothDir = = bothDir
```

and a number of observer axioms (not shown here). A segment has a registration of the form *train(t)* if a train t has registered for that segment, and *no* otherwise. A segment direction is either one of the global directions (*dirAB* or *dirBA*) or is bidirectional (*bothDir*).

The idea is, that a train must only be allowed to move to a segment if it is registered on that segment and if its direction is consistent with the direction of that segment.

Hence, the requirements to the guard *can_move* is refined to,

```
axiom /∗ requirements to guard can_move ∗/
  [can_move_implication2]
    ∀ σ : State, t : TrainId •
      consistent(σ) ∧ can_move(σ, t) ⇒
        case position(σ, t) of
          single(s1) →
            let dir = direction(σ, t) in
              (∃ s2 : Segment •
                are_connected(σ, to_end(s1, dir),
                from_end(s2, dir)) ∧
                registered(σ, s2) = train(t) ∧
                seg_dir(σ, s2) ∈ {dir, bothDir}
              )
            end,
          double(s1, s2) → true,
          _ → false
        end
```

and the state invariant must imply that any train is registered on and has a direction which is consistent with the segments of its position.

axiom /∗ additional requirements to consistent ∗/
[consistent_implication2]
∀ σ : State • consistent(σ) ⇒
(∀ t : TrainId, s : Segment •
s ∈ segments(position(σ, t)) ⇒
registered(σ, s) = train(t) ∧
seg_dir(σ, s) ∈ {direction(σ, t), bothDir}
)

Third Specification. In the third specification, segment reservations for trains at switch boxes is introduced, and segment registrations is defined in terms of that. Furthermore, a concept of locking of points is introduced. The idea is that a train must lock a point in order to pass it, and when a train has locked a point, the point cannot be switched before the train has passed the point.

This gives rise to new state observers for observing which reservations and locks have been made,

value /∗ state observers ∗/
reserved : State × SwitchBoxId × Segment →
Register,
locked : State × PointId → Register

and new state constructors with guards for making reservations and locks, new observer axioms, axioms about the new guards, refinement of the *[can_move_implication]* and *[can_switch_implication]* axioms (using the new concepts), and additional requirements to the state invariant.

While the actions of making a reservation or lock are events independent of the move event, the deletion of reservations and locks are part of the move event: The new observer axioms for the move constructor express that when a train leaves a segment and passes a switch box its reservations at this switch box will be removed, and when it passes a point the lock of the point will be removed.

Fourth Specification. In the fourth specification, a notion of train routes is introduced and sensors at the switch boxes sense when trains are passing.

This gives rise to new observers:

value /∗ state observers ∗/
segments_of_route : State × TrainId → Segment*,
sbs_of_route : State × TrainId → SwitchBoxId*,
sensor : State × SwitchBoxId → **Bool**

The two first observers give the list of segments and switch boxes along the route of a train. For a state to be consistent, it is now also required that these two lists constitute a route wrt. the given network and conform with the direction and position of the train (the train should be positioned at the start of the route and have a direction towards the end of the route). New observer axioms (not shown here), for the move constructor, express that when a train leaves a segment or passes a switch box, these are removed from the list of route segments and switch box segments, respectively.

The third observer gives true for a switch box if and only if a train is passing the switch box. The values returned by this observer must, by physical law, conform with (and are actually derivable from) the *position* observer. This is expressed in an axiom not shown here.

5.3 Fifth Specification

Finally, in the fifth specification, we are able to define a concrete state space consisting of a state space for each train and a state space for each switch box,

type
State = {| σ : State' • is_wff(σ) |},
State' = (TrainId $\xrightarrow[m]{}$ TrainState) × (SwitchboxId $\xrightarrow[m]{}$ SwitchboxState),
TrainState :: ...,
SwitchboxState ::
connected : SegmentEnd
locked_by : Register
sensor : **Bool**
res_map : Segment $\xrightarrow[m]{}$ Register
seg_dir_map : Segment $\xrightarrow[m]{}$ SegDir
in_action : Register

where *TrainState* and *SwitchboxState* are given explicit formal representations (record types) for the local train state and switch box state, respectively. These representations correspond to the informal descriptions in Fig. 3. We only consider states (defined by a predicate *is_wff*) which satisfy the axioms of physical laws (like "only neighboring segments are connected") of the model.

With this explicit definition of *State*, it is now possible to replace all axioms with explicit function definitions in terms of functions defined for the two new types *TrainState* and *SwitchboxState*. For instance, the observer function *direction* can be defined as follows:

direction : State × TrainId → Direction
direction((σ_t, σ_s), t) ≡ direction(σ_t(t))

where *direction* is an observer function defined for train states (of type TrainState), and the state invariant can be given a definition of the form:

consistent : State → **Bool**
consistent(σ) ≡ p1(σ) ∧ ... ∧ p5(σ)

5.4 Verification

Implementation Relations. In each of the development steps (from specification number i to specification number $i+1$, $i = 1, ..., 4$) above, we have used the RAISE justification tools to prove that the new specification is a *refinement* of the previous specification, i.e., the new specification provides declarations of at least all the types and functions provided by the previous specification, and all the axioms of the previous specification are consequences of the axioms of the new specification.

For instance, for the development step from the first to the second specification, the refinement proof amounts to prove that *[can_move_implication1]* is a consequence of *[can_move_implication2]* and the other axioms in the second specification. The proof of this can be found in an electronic archive [7].

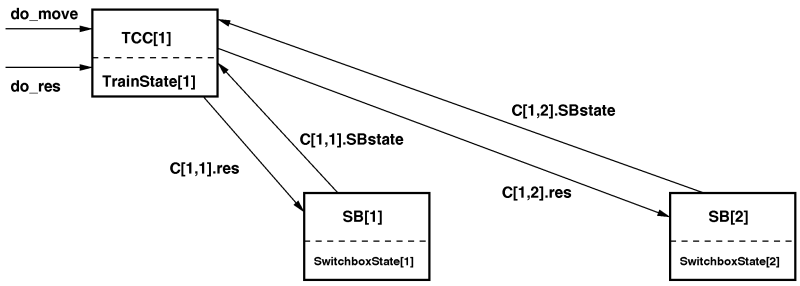

Fig. 4. Distributed architecure with train control computers, switch boxes and communication channels.

Satisfaction of Safety Requirements. For each of the first four specifications, we prove that it is consistent with the strong safety requirements stated in the beginning of this section, and finally for the fifth specification we prove that it fully satisfies these requirements.

The *[consistent_is_safe]* theory is verified to hold already for the first specification. Then, since refinements preserve theories, we know that it also holds for the second to fifth specification.

Verification of the *[can_con]* theories is done stepwise: For specification number i we prove,

$$\forall \ldots \bullet \ \text{consistent}(\sigma) \wedge \text{can\_con}(\sigma, \ldots) \Rightarrow \text{pi}(\text{con}(\sigma, \ldots))$$

For instance, for the first specification, one of the theories we prove (see Appendix) is,

[safe_can_move]
$\forall\ \sigma$: State, t : TrainId •
consistent(σ) $\wedge$ can_move(σ, t) $\Rightarrow$ safe(move(σ, t))

Then, since refinements preserve theories, the fifth specification satisfies,

$$\forall \ldots \bullet \ \text{consistent}(\sigma) \wedge \text{can\_con}(\sigma, \ldots) \Rightarrow (\text{p1}(\text{con}(\sigma, \ldots)) \wedge \ldots \wedge \text{p5}(\text{con}(\sigma, \ldots)))$$

which is equivalent to the *[can_con]* theory, the definition of *consistent* in the fifth specification.

Verification of the *[can_con1_con2]* theories is done similarly.

6 Development of the Railway Control System: Second Stage

The fifth specification presented above introduced explicit implementable states for trains and switch boxes. However, at that stage no architectural requirements were present, so that different centralized or distributed system designs may be elaborated as correct implementations of this specification. The second stage of our development introduces a concrete architectural design and communication protocol for a distributed railway controller consisting of concurrent communicating processes,

value
controller : State $\xrightarrow{\sim}$ **in any out any Unit**
controller(σ_t, σ_s) ≡
(‖ { TCC[t].main(σ_t(t)) | t : TrainId})
‖
(‖ { SB[s].main(σ_s(s)) | s : SwitchboxId})

where $TCC[t].main(\sigma_t(t))$ is a process representing the train control computer in train t, and $SB[s].main(\sigma_s(s))$ is a process representing switch box s. These processes are defined in terms of the guards, state constructors, and observers defined in the first major stage, and follow the protocol described in Section 2. The **in any out any** construct, in the signature of the definition of the controller process, is an access description stating that the process is allowed to communicate on any channel, and the result type **Unit** indicates that the process does not return any value.

The transition from the last specification stage to the distributed design stage is performed according to a standardized procedure resulting in designs which are consistent to the specification in a natural way (cf. Fig. 4).

- The global specification state is mapped in one-one correspondence to the distributed components. For global state (σ_t, σ_s), train tid and switch box bid, σ_t(tid) is mapped to TrainState[tid] and σ_s(bid) is mapped to SwitchboxState[bid].
- Application of each constructor *con* on a train state and/or a switch box state is guarded by a channel command and the corresponding *can_con* guard defined in the fifth specification layer. Observe that the train and switch box state spaces have been designed in such a way that each guard evaluation can be based on the local state space only. For example, a train control computer will allow the train to move if it is triggered by the do_move channel and the *can_move* guard evaluates to $true$ on the local state space.
- For correct implementation of the fifth specification layer, corresponding state components in trains and switch boxes (for example, the reservation state and the lock state described in Section 2) must be consistent, whenever a guard using this state information is evaluated. To ensure this, a communication protocol between trains and switch boxes is designed to implement the reservation constructor introduced in the specifications: Train tid sends a reservation request on channel C[tid,bid].res to switch box bid. The switch box evaluates a local guard and responds by returning its possibly updated state space to the train via channel C[tid,bid].SBstate. This information is used by the TCC to update its local information about reservations and locks.

7 Discussion

7.1 Lower-Level Safety Requirements Related to Real-Time Behavior

From untimed safety requirements to real-time requirements... The formal models used in the sections above do not incorporate the notion of time. This is motivated by the fact that the top-level safety requirements, described in Section 4, were of an untimed nature or, more precisely speaking, should hold unconditionally for all points in time.

We recommend that in these situations untimed formalisms should be used, since the abstraction from time considerably simplifies the specification and verification process. However, further refinements of the abstract models may require the introduction of time, so that other specification languages and associated proof methods have to be used. This is shown by the following example referring to the can_move-predicate introduced in the specifications above.

The control algorithms above guarantee safety if trains do not enter their next segment as long as the can_move predicate evaluates to $false$. In order to implement this condition, trains must trigger a braking process before the end of the segment they are driving on, so that they come to a stop before entering the next segment. The last possible point p_{brake} on the track segment to trigger the brakes depends on the type of the train,[6] its speed and its distance from the end of the actual segment. This is a safety condition which can only be expressed using timed (hybrid) specification formalisms. However, if the calculation of p_{brake} is available as a function of train type and speed, and if the train's speed and position can be determined in a reliable way, the associated control task is very simple; it says, "*trigger the braking process, whenever p_{brake} is reached and the can_move condition to enter the next segment still evaluates to false.*"

In this example, the advantages of a separation of concerns with respect to untimed and timed behavioral aspects become apparent.

- The specifications presented in the preceding sections were complex because they dealt with *multiple* trains, switch boxes and segments and the *interactions* between these objects; being able to abstract from timing aspects was advantageous in order to keep the complexity manageable.
- The time-related safety conditions sketched in the paragraph above deal with a *single* train and its distance from the next switch box. The control components enforcing these conditions do not have to "know" anything about segments, directions, point states, or about other trains. This facilitates the development and verification of the time-related control task in a considerable way.

...and back to untimed safety requirements. An additional consideration related to the *dependability* of a safety mechanism implementation can lead to additional lower-level safety requirements: In practice, the implementation of the automatic braking mechanism is not considered as sufficiently dependable, and recent railway accidents in Germany support this assumption. As a consequence, the allocation of shunts and the enforcement of flank protection, mentioned in Section 4, are introduced as additional safety requirements which are "orthogonal" to the time-related control tasks sketched above. According to our understanding, these safety requirements are really implementation-dependent and should not occur on the top-level, specified in Section 4, if the automatic position- and speed-dependent braking mechanism could be implemented in a sufficiently dependable way, the requirements related to shunts and flank protection could be dropped.

Observe that shunts and flank protection again represent untimed safety requirements, so it cannot be said in general that more abstract safety requirements were untimed, while implementation-dependent ones would mostly be related to timing aspects.

Top-level safety requirements may be inherently time-dependent. It is not always the case that top-level safety requirements can be defined without using a notion of time. For example, according to German safety regulations, a railway crossing is considered as unsafe if the road traffic has been blocked by red traffic lights for more than a certain time limit. After this time limit has been reached, it must be taken into account that car drivers and pedestrians might no longer observe the red lights and cross the tracks without permission. Such a safety condition is inherently time-related, it cannot be abstracted to conditions about observable event occurrences. In such cases, timed specification formalisms have to be used already at the most abstract modeling stages.

7.2 Alternatives for the Engineering Concept

The control mechanisms introduced in this article refer to segments and switch boxes which have static locations in the railway network. In the context of high-speed trains, other approaches are currently being considered [cf. 2]. The no-collision safety requirements, introduced in Section 4, can also be modeled using the concept of an *envelope*. This is a dynamically determined neighborhood of the train depending on its position and speed. Safety is enforced by the requirement that the envelopes of different trains should never overlap. This allows to drop the concept of static segments allocated to trains, but is more complex to implement in a dependable way. Therefore, the concepts described in our article cannot be transferred to the concept of envelopes in a direct way.

7.3 Future Work: Frameworks and Generic Theories for Railway Control Systems

The distributed railway control system introduced in this article is already generic with respect to network size and number of trains. However, the train control computer specifications $TCC[t].main(\sigma_t(t))$, sketched in Section 6, still operate explicitly on the state information $C[i,j].SBstate$ received from the switch boxes that are involved in the actual reservation and locking requests. As a consequence, $TCC[t].main(\sigma_t(t))$ would have to be completely redesigned as soon as the structure of $C[i,j].SBstate$ would change due to technical reasons or as soon as a new type of

6. The braking pressure has to be applied according to a pressure-over-time curve depending on the train type.

safety-critical object—for example, a railway crossing—had to be introduced in the network.

To reduce the development and verification costs for such control systems, we advocate to investigate a more generic design for the train control computer which abstracts from the specific type of switch boxes guarding the critical places in the railway network. This abstraction introduces *points of control (POC)* which are locations in the railway network that may only be passed by a train if it has the proper *permissions* associated with each point. Each POC is associated with a set of *controllers*, these are objects managing all the information necessary to grant or reject a permission request according to the present system state. If a centralized control concept is implemented, we have just one controller—the centralized interlocking system—for all POCs in the network. In the distributed approach described in this article, controllers are implemented as switch boxes managing specific units in the railway network. Typically, a concrete instance of a POC may be,

1. A point separating two track segments: Here, the controllers consist of the switch boxes associated with the point as described in the previous sections. The permission condition is "instantiated" by the reservations and locks to be attained from the associated switch boxes as elaborated in the previous sections.
2. The boundary between two blocks on the track, without a separating point: Here a similar type of switch boxes may be used for controllers, as in 1, and the permission may be defined in a similar way, but now it is not necessary to attain locks since there are no points present,
3. A railway crossing guarded by traffic lights and possibly barriers. This POC can be managed by a single switch box of another type than the ones used in 1 and 2. It switches traffic lights and opens or closes barriers. The permission is granted if the crossing is safe due to activated redlights and lowered barriers.

On this abstract level, the train control computer operates using the following generic control algorithm involving generic operations and functions *Request_permission*, *Collect_permission_data*, p_{brake}, and *can_move*.

1. *Request_permission* for the next POC to be passed. This operation uses a generic function mapping the POC and the train position[7] to the set of controllers which have to be addressed to get a permission. It sends a request telegram to each of these controllers. Again, telegram transmission is a generic suboperation, because the transmission media and protocols to be used will depend on the type of controllers to be addressed.
2. *Collect_permission_data* from the controllers determined in Step 1.
3. As soon as p_{brake} has been reached, evaluate *can_move* using the collected permission data. On the generic level, *can_move* traverses the list of controllers determined in Step 1 and calls a *permission_is_granted* function to evaluate the permission data received from each controller in Step 2. *Can_move* returns $true$ if each call to *permission_is_granted* returns $true$.
4. If *can_move* evaluates to $true$, the train may move on, otherwise the breaking process is triggered.

Interpreting this approach from the viewpoint of object-orientation, we are constructing a *framework* in the sense of [4] for distributed railway control systems. Frameworks are generic collections of cooperating classes associated with a reusable design structure. By means of instantiation, subclassing and composition of class instances, frameworks are customised for a specific application context. Our framework includes generic classes for railway networks, trains (represented by their control computers), and points of control in the network. The collaboration between instances of these classes is defined by the protocol sketched above which requires that trains may only pass the next POC as long as they have the associated permission.

An important advantage of frameworks which has not been sufficiently considered in [4], consists in the fact that frameworks may be associated with *generic theories*, that is, collections of generic theorems which may be instantiated together with the framework, resulting in concrete theorems about correctness or safety properties of the concrete system. The safety properties established in the previous sections for the control algorithms introduced represent theories which are generic with respect to the railway network and the number of trains involved. The approach sketched in the paragraphs above leads to even more powerful theories which are also generic in the type of POC and in the state spaces associated with the related switch boxes. The main advantage of generic theories is given by the fact that the associated theorems have been established on abstract generic level and are therefore inherited by every concrete instance of the framework. This increases the efficiency of the verification process in a considerable way (see [8] for a report on applications in the context of software verification for space mission systems).

Further examples of this framework-oriented approach for the development and verification of railway control systems are given in [9]. Currently, we plan to elaborate a collection of frameworks specialized for applications in the context of railway control systems.

7.4 Conclusion

In this article, we have presented the engineering concept and the design and verification of a control algorithm for a distributed railway control system. Typically, the concept would be applied for small local railway networks which have been formally classified as *simple networks* in this article. It should be emphasized, however, that other (and more expensive!) approaches are required in the case of large high-speed networks, where safety-related concepts cannot be simply expressed by means of the isolated points of control we have used in the present paper. We consider

7. Recall that the set of switch boxes to be addressed in order to pass a point depends on the direction the train is approaching this point. Other POCs—for example, railway crossings—are independent of the side from which they are approached.

the following aspects of our work to be the main advantages in comparison to other work that has been performed in the field of design and verification of similar systems (see [2] as an example of another practically relevant approach to formal specification and verification in the railway domain).

- Our refinement approach, starting with highly abstract algebraic specifications and ending with concrete distributed programs, helps to separate general aspects of train control mechanisms and their safety from concrete application-specific design decisions.
- Our verification concept is independent on the size of the underlying network topology. In contrast to that, experiments with model checking have led to unmanageable explosions of the state space, as soon as more complex networks were involved or a larger number of trains had to be controlled.
- Within the restrictions of the *simple network* definition given above, the network topologies covered by our algorithm are fairly general. There are no limits regarding the size of the network, the number tracks involved, or the places where points may occur. In contrast to that, approaches using compositional reasoning and structural induction over the underlying network topologies only seem to work for unrealistically simplified networks.
- Starting with a most abstract version of safety requirements, our approach allows to verify their completeness and trace their "implementation" in the more concrete refinements of the abstract control algorithm in a straight-forward manner. For approaches defining only implementation-specific safety requirements without reference to a more abstract safety concept, it is nearly infeasible to check safety requirements with respect to completeness.

We would like to emphasise that the control algorithm presented here represents just a building block in a more general approach for the development, verification, validation, and test (VVT) of safety-critical systems which is investigated by the authors' research groups at DTU and the University of Bremen. In this wider context, our research work covers:

- A systems engineering approach for safety-critical systems which is driven by hazard analysis, risk analysis, and a design approach taking VVT issues into consideration right from the beginning of the development life cycle.
- Improvement of the development and VVT process for safety-critical systems by utilization of design patterns, frameworks, and associated generic theories.
- Software-architectures for safety controllers.
- Automated real-time testing for embedded hardware/software components.
- An integrated standardized concept for verification, validation, and test of safety-critical embedded controllers, applying combinations of VVT methods, each one optimized for a specific step in the system development life cycle.

APPENDIX

PROOFS

The verification of the proof obligations mentioned in Section 5.4, has been done using the RAISE tools. Below, we give a short introduction to RAISE verifications and sketch a proof of a safety theory *[safe_move]* for the first specification. For more details about RAISE verifications, see [11].

In RAISE, the verification of a proof obligation is done by a series of steps in which proof rules are applied transforming the goal (proof obligation) into new goals the truth of which ensures the truth of the original goal. There are two kinds of proof rules, equivalence rules, and inference rules. Equivalence rules are used to replace subterms of a goal by equivalent subterms. For instance, the following equivalence rule:

[and_implies] eb ∧ eb′ ⇒ eb′′ ≅ eb ⇒ eb′ ⇒ eb′′

states that any subterm of the form $eb \wedge eb' \Rightarrow eb''$ can be replaced with a subterm of the form $eb \Rightarrow eb' \Rightarrow eb''$, and vice versa. Inference rules are used to replace a goal with other goals. For instance, the inference rule,

[imply_deduction_inf1]
$$\frac{[id]ro\_eb \vdash ro\_eb\prime}{ro\_eb \Rightarrow ro\_eb\prime}$$
when convergent(ro_eb) ∧ **pure**(ro_eb)

states that a goal of the form $ro\_eb \Rightarrow ro\_eb\prime$ can be proved by proving $ro\_eb\prime$ in a context where the axiom $[id]ro\_eb$ is assumed to be true, if ro_eb is convergent (i.e., terminates) and pure (i.e., does not contain variables).

Below, we show the first steps of a verification of the safety theory *[safe_move]* (given in Section 5.4) for the first specification (given in Section 5.1). The verification is done in the *context* of this first specification, which means that we may use the axioms and definitions of that specification to prove our goal.

⌊∀ σ : State, t : TrainId • consistent(σ) ∧ can_move(σ, t) → safe(move(σ, t))⌋
all_assumption_inf:
⌊consistent(σ) ∧ can_move(σ, t) ⇒ safe(move(σ, t))⌋
and_implies :
⌊consistent(σ) ⇒ can_move(σ, t) ⇒ safe(move(σ, t))⌋
imply_deduction_inf1 :
[consistent] consistent(σ) ⊢
⌊can_move(σ, t) ⇒ safe(move(σ, t))⌋
imply_deduction_inf1 :
[can_move] can_move(σ, t) ⊢
⌊safe(move(σ, t))⌋
unfold_safe:
⌊no_collision(move(σ, t)) ∧ no_derailing(move(σ, t))⌋
and_split_inf :

- ⌊no_collision(move(σ, t))⌋
  /∗ proof of sub-goal ∗/ ...

- ⌊no_derailing(move(σ, t))⌋
  /∗ proof of sub-goal ∗/ ...

In the proof, the goals are enclosed by brackets ⌊ and ⌋, and the names of the applied proof rules are written between the goals. In the first step we assume that we have

fixed, but arbitrary state σ and train t. In the second step, the above shown equivalence rule *[and_implies]* is applied to the whole goal. In the third and fourth step, we apply the inference rule [*imply_deduction_inf1*] obtaining a new goal which must be proved in the context of the two assumptions named [*consistent*] and [*can_move*]. In the fifth step, we unfold the application of the function $safe$ using the definition of $safe$ (see Section 4). Then, in order to structure the specification nicely, we use an inference rule to split our goal into two smaller subgoals.

The first subgoal is (after a few manipulations) proved by a case analysis with three cases:

```
⌊no_collision(move(σ, t))⌋
unfold_no_collision, all_assumption_inf,
imply_deduction_inf1 :
  [distinct]  t1 ≠ t2 ⊢
    ⌊segments(position(move(σ, t), t1)) inter segments
    (position(move(σ, t), t2)) = {}⌋
  cases
  [case1]  t ≠ t1 ∧ t ≠ t2 ⊢
    ⌊segments(position(move(σ, t), t1)) ∩
    segments(position(move(σ, t), t2)) = {}⌋
    /* proof of case1 */ ...
  [case2]  t = t1 ⊢
    ⌊segments(position(move(σ, t), t1)) ∩
    segments(position(move(σ, t), t2)) = {}⌋
    /* proof of case2 */ ...
  [case3]  t = t2 ⊢
    ⌊segments(position(move(σ, t), t1)) ∩
    segments(position(move(σ, t), t2)) = {}⌋
    /* proof of case3 */ ...
```

Here, we show the proof of the first case:

```
  ⌊segments(position(move(σ, t), t1)) ∩
  segments(position(move(σ, t), t2)) = {}⌋
position_move :
  ⌊segments(if t1 = t then next_position(σ, position(σ, t1),
          direction(σ, t1))else position(σ, t1) end)
    ∩ segments(position(move(σ, t), t2)) = {}⌋
  since
      ⌊safe(σ)⌋
    imply_modus_ponens_inf :
    • ⌊consistent(σ)⌋
      consistent :
        ⌊true⌋
      qed
    • ⌊consistent(σ) ⇒ safe(σ)⌋
      consistent_implication1 :
        ⌊true⌋
      qed
  end
simplify :
  ⌊segments(position(σ, t1)) ∩ segments(position(move
  (σ, t), t2)) = {}⌋
position_move, simplify :
  ⌊segments(position(σ, t1)) ∩ segments(position(σ, t2)) =
  {}⌋
no_collision :
  ⌊true⌋
  since
  • ⌊t1 ≠ t2⌋
    distinct :
      ⌊true⌋
    qed
  • ⌊no_collision(σ)⌋
    imply_modus_ponens_inf :
    • ⌊safe(σ)⌋
      /* proved as above */ ...
    • ⌊safe(σ) ⇒ no_collision(σ)⌋
      unfold_safe :
        ⌊no_collision(σ) ∧ no_derailing(σ) ⇒
        no_collision(σ)⌋
      simplify :
        ⌊true⌋
      qed
  end
qed
```

In the first step, the *[position_move]* axiom (see Section 3.2) has been applied. Since this axiom has as precondition $safe(\sigma)$, we have to prove that as well on the side after the keyword **since**. This is done using the [*consistent*] assumption and the axiom [*consistent_implication1*] given in the first specification (in Section 5.1). In the second step, a simplier tool has used the *[case1]* assumption to simplify the if-expression further. Then follows two steps similar to the two first ones. And finally, the goal is reduced to **true** by using the definition of the *no_collision* function and proving two goals on the side. This completes the proof of first case of the case analysis.

The proof (not shown here) of the second case is done by a case analyse over the possible positions of the train t. The proof of the third case is symmetric to that for the second case.

The full proof can be found in an electronic archive [7].

REFERENCES

[1] D. Bjørner, C.W. George, B. Stig Hansen, H. Laustrup, and S. Prehn, "A Railway System, Coordination '97, Case Study Workshop Example," Technical Report 93, United Nations University's Int'l Inst. for Software Technology, Macau, 1997.

[2] B. Dehbonei and F. Mejia, "Formal Development of Safety-Critical Software Systems in Railway Signalling," *Applications of Formal Methods*, M.G. Hinchey and J.P. Bowen, eds., pp. 227–252, Prentice Hall, 1995.

[3] U. Oser, J. Arms, and H. Wegel, "FunkFahrBetrieb (FFB) Zum Wirtschaftlichen Einsatz auf Regionalstrecken," Eisenbahntechnische Rundschau (ETR) vol. 46 Heft 6, pp. 323-331, 1997.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns.* Addison Wesley, 1995.

[5] K. Mark Hansen, "Linking Safety Analysis to Safety Requirements—Exemplified by Railway Interlocking Systems," PhD dissertation, Technical Univ. Denmark, Lyngby, Dept. of Information Technology, 1996.

[6] K. Mark Hansen, "Formalising Railway Interlocking Systems," *Proc. Second FMERail Workshop*, Oct. 1998.

[7] A. Haxthausen and J. Peleska, "A Distributed Railway Control System–Selected Proofs," Feb. 2000. http://www.it.dtu.dk/~ah/Dracos.

[8] J. Peleska and B. Buth, "Formal Methods for the International Space Station ISS," *Correct System Design.* E.-R. Olderog, and B. Steffen, eds., pp. 363-389, 1999.
[9] J. Peleska, A. Baer, and A. Haxthausen, "Towards Domain-Specific Formal Specification Languages for Railway Control Systems," *Proc. Transportation Systems 2000,* June, 2000.
[10] The RAISE Language Group, *The RAISE Specification Language.* The BCS Practitioners Series, Prentice Hall, 1992.
[11] The RAISE Method Group, *The RAISE Development Method.* The BCS Practitioners Series, Prentice Hall, 1995.
[12] N. Storey, *Safety-Critical Computer Systems.* Addison Wesley, 1996.

Anne Haxthausen received the Ph.d. degree from the Technical University of Denmark in 1989. She is associate professor at the Department of Information Technology, Technical University of Denmark. From 1988 to 1994, she was employed at Dansk Datamatik Center and CRI A/S in Denmark, where she was a key person in the ESPRIT projects RAISE (Rigorous Approach to Industrial Software Engineering) and LaCoS (Large-scale Correct Software using formal methods). Together with colleagues, she developed RAISE: a mathematically-based software development method with an associated formal specification language and tools. Furthermore, she was active in the technology transfer of RAISE to industry and universities. In 1993, she was guest researcher at Electrotechnical Laboratory in Japan. Since 1995, she has been associated with the Technical University of Denmark, where she is doing research in formal methods for software development. Her research interests span from the development of such methods and their mathematical foundations to their practical industrial application. Current industrial applications of her research work focus on the development and verification of safety critical railway control systems.

Jan Peleska is professor for computer science (operating systems and distributed systems) at Bremen University in Germany. From 1984 to 1990, he worked with Philips as senior software designer and later on as department manager in the field of fault-tolerant systems, distributed systems, and database systems. From 1990 to 1994, he was manager of a department at Deutsche System-Technik responsible for the development of safety-critical embedded systems. Since 1994, he has worked as a consultant, specialising on development methods, verification, validation, and test of safety-critical systems. He is cofounder of Verified Systems International GmbH, a company providing tools and services in the field of safety-critical system development, verification, validation and test. His current research interests include formal methods for the development of dependable systems, test automation based on formal methods with applications to embedded real-time systems, verification of security properties, and formal methods in combination with CASE methods. Current industrial applications of his research work focus on the development and verification of avionic software, space systems, and railway control systems. He is associated with the Bremen Institute of Safe Systems (BISS), a division of the Center for Computing Technology TZI at the Department of Mathematics and Computer Science of Bremen University.