SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 01-42

Model Checking Railway Interlocking Systems

Kirsten Winter

Version 1, December 2001
Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

To appear in *Proceedings of Australian Computer Science Conference 2002*.

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

Model Checking Railway Interlocking Systems

Kirsten Winter

Software Verification Research Centre
University of Queensland,
Queensland 4072 Australia
Email: kirsten@svrc.uq.edu.au

Abstract

For supporting the analysis of railway interlocking systems in the early stage of their design we propose the use of model checking. We investigate the use of the formal modelling language CSP and the corresponding model checker FDR. In this paper, we describe the basics of this formalism and introduce our formal model of a railway interlocking system. Checking this model against the given safety requirements, the signalling principles, we get useful counter-examples that help to debug the given interlocking design. This work provides a successful example of how formal methods can be used to support the industrial development process.

Keywords: Formal methods, model checking, interlocking systems, process algebras.

1 Introduction

The development of railway signalling systems is currently very labour intensive and requires specialised skills. To reduce the number of possible errors in the process, Queensland Rail (QR), the major railway operator and owner in Queensland, Australia, intends to support its design process by a specialised tool set, which we call the Signalling Design Toolset (SDT). A general architecture of this design process and the toolset is introduced in [RBK+01].

The signalling design process involves the following documents:

- the *track-layout* (or signalling layout), which defines the position of signals and points in a particular section of the railway system and the permitted routes between the signals;
- a functional specification for the signalling of the given track-layout; this is currently given in form of a so called *Control Table* and describes how the general *Signalling Principles*, which describe the safety requirements, are realised in this particular track-layout;
- an implementation of the Control Table in either software or electrical relays which is called the *Interlocking*.

In order to guarantee safety of a signalling system QR manually validates the Interlocking against the Signalling Principles. This task is labour intensive and prone to error. Moreover, possible errors in the design are detected very late in the design process. To overcome these problems we are aiming at fully automatic tool support for verification that can be used in the early stages of design. We propose to integrate formal methods into the design process and to use *model checking* to ensure the correctness of a design before it is implemented. This provides an example of how industry can benefit from using formal methods during their development process.

The aim of this paper is to describe a formal model of the functional specification for a track-layout that can be checked automatically against a formal model of the Signalling Principles. We call the formal model of the functionality of the interlocking the *Interlocking Model*. The formal model of the Signalling Principles is called the *Principle Model*. Our aim is to translate the Control Tables into a minimal Interlocking model, which we then model check against the Principles Model. Our approach, however, is limited in the size of the model since model checking involves a complete search of the state space. For signalling systems, this restriction implies that the treated track-layout must not be too big.

As a formal modelling language we use, as a first attempt, the language of Communicating Sequential Processes (CSP) ([Ros98]) and as the corresponding model checker FDR ([Ros94]). CSP and the FDR model checker are briefly introduced in the next section. Section 3 describes the functionality of an interlocking system that we want to formalise. The formal Interlocking Model and the Principles Model is given in Section 4. Checking the Interlocking Model against the Principles Model leads to counter-examples that help to debug the Interlocking Model. We give some examples in Section 5 and conclude this paper with a discussion in 6.

2 The formal modelling language CSP

In this section, we briefly describe the formalism CSP[1] to the reader who is not familiar with it. Naturally, this does not provide a full definition of this language but is restricted to the main features that we used in our model.

In CSP, a system is typically described in terms of processes that interact or communicate. These processes are characterised through their *behaviour*. Since the language has no notion of a system's state the behaviour is given as a set of possible sequences of atomic *actions* (events) that can be executed (occur). We describe a single process `P` through a list of choices for the next action to take. After an action is taken the process starts choosing again. This is specified in CSP in the following way:

[1] The syntax that we use here is the syntax of CSP^M, which is the machine-readable derivation of CSP introduced in [Ros98].

Copyright ©2001, Australian Computer Society, Inc. This paper appeared at the Twenty-Fifth Australasian Computer Science Conference (ACSC2002), Melbourne, Australia. Conferences in Research and Practice in Information Technology, Vol. 4. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

```
P = a -> P
 [] b -> P
 [] c -> P
```

The choice is marked by the `[]` operator and an action followed by something else is described through an expression of the form `a -> P`. The execution of an action can be guarded. The expression

```
guard & a -> P
```

describes a simple "if-then-else" construct: if the Boolean expression `guard` is true then the process executes action `a` and proceeds with process `P` otherwise - if `guard` is not true - it terminates which is modelled by the process `STOP`.

Regarding the options for a choice of actions it is important to understand that one *possible* choice has to be taken (i.e., the guard has to be satisfied in the current situation) but apart from this the choice is free. This is given through the semantics of CSP and allows a broad variety of possible behaviour to be modelled.

There are several ways to compose small processes into a single big process. The small processes may run at the same time independently without interacting or they may depend on each other in that they have to interact (communicate) when taking actions.

The expression

```
P ||| Q
```

describes a process that consists of sub-processes `P` and `Q` which run *independently* in parallel to each other; they can execute their actions at the same time and do not influence each others' choice. The generalised expression

```
||| id:ID @ P(id)
```

describes the same situation for a number of processes; the name `P(id)` denotes a parameterised process where `id` is taken from a set of indices `ID`.

The expression

```
P [| a,b |] Q
```

describes a process that consists on two sub-processes `P` and `Q` that run in parallel, however, they have to communicate over the actions `a` and `b`. That is, `P` can only perform action `a` or `b`, if the process `Q` performs the same action at the same time. This is a strong means for restricting the behaviour of a process. For instance, if `Q` is a process that executes `a` and `b` alternatingly then `P` is forced to alternate these actions as well; or if `P` does not contain the choice of action `a` at all then `Q` cannot execute `a` either since `P` can never agree on this action.

The generalised expression

```
[| a, b |] id:ID @ P(id)
```

puts a number of processes `P(id)` in parallel such that they all have to agree on the actions `a` and `b`.

Actions (or what is called *events* in the literature) may contain data values as well. The corresponding language construct is called a *channel*. This notion refers to the idea of message passing between processes. The expression `move.from.to` denotes a channel `move` that contains as data values the information of the current location `from` and the next location `to` the process wants to move to. `from` and `to` can be instantiated with corresponding values, e.g., `move.office.home` describes the movement from the office to home. The expression `{| move |}` describes the set of movements between all possible locations in the system; `from` and `to` are *unfolded* to every possible value, i.e., they are instantiated with every possible location. Given this feature we can model the following

```
[| {| move |} |] id:Person @ Marching(id)
```

This expression describes a number of marching persons that all have to move at the same time from and to the same location. If we want to loosen the restriction of movement for these people, e.g., they should only move together along a route that starts at the central station, we can limit the unfolding of values to those of the last parameter `to` only:

```
[| {| move.central_station|} |] id:Person
                          @ Marching(id)
```

In this expression, the movement from any point other than `central_station` can be done individually.

The FDR tool implements an algorithm for checking *refinement* between two given processes: One of these processes describes our Principle Model, the other our Interlocking Model. Between both models a refinement relation is given if every possible behaviour of the Interlocking Model is also a possible behaviour of the Principle Model. That is, if the Interlocking Model allows behaviour that is *not* allowed by the Principle Model, this indicates that the Signalling Principles are violated by the Interlocking Model. In this case, a *counter-example* is output by the FDR tool, which shows the erroneous behaviour of the Interlocking Model.

3 The functionality of an interlocking system

The signalling functions controlling train movements at a particular location are known as an interlocking (because setting one movement locks out other movements). It is typically defined in terms of *routes* from one signal to another. A route is divided into several tracks. In Figure 1, for instance, the route from signal `s12` to `s8` (called `r12_1m`) comprises the tracks `tad and tae`. The track `taz` is called the *overlap* of route `r12_1m`, i.e., the track after the signal that guards the next route.

The interlocking specifies the necessary conditions to *clear* a route, which means to prepare a route such that it can be used by a train. Example conditions include

- the points in the route must be set to the right direction and locked (so that they cannot be moved).
- the *tracks* of the route must be proved clear of trains.

If these conditions are true, the entry signal of the route can be cleared (i.e., show green or yellow). Once cleared, a route is locked by an approaching train. This approach-locking mechanism can be released after the route is used by the train or by timing the train to ensure it is stationary.

In our work, we use a particular track-layout as a case study which we call the "MiniAlvey". It is a reduced version of the layout used in [SWD97]. Figure 1 depicts the track-layout. It is a circular layout that comprises six tracks, two points and three signals. We distinguish four routes in this track-layout, namely route `r12_1m` from signal `s12` to signal `s8`, route `r14_1m` from signal `s14` to signal `s8`, route `r8_1m` from signal `s8` to signal `s12`, and route `r8_2m` from signal `s8` to signal `s14`. We restrict the usage of this layout to the clockwise direction only. A train might run from the track `tab` to the upper track `tba` or to the lower track `tac`.

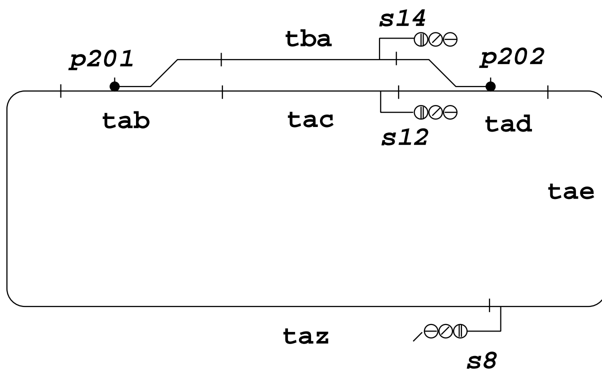

Figure 1: The "MiniAlvey" track-layout

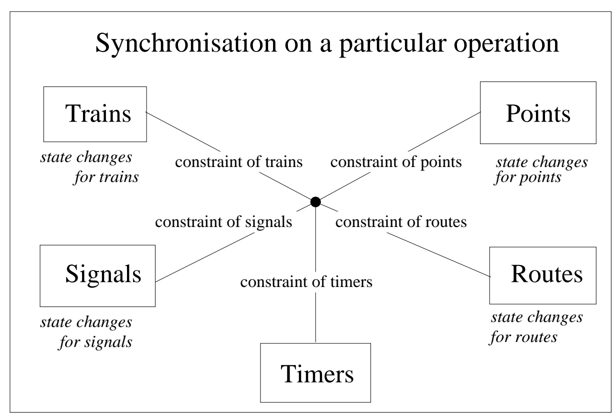

Figure 2: Process synchronisation in the CSP model

4 The Formal Models

Signalling Principles define the safety properties of an interlocking system. However, in most cases these safety properties do not take into account the reachability of system states. That is, no matter if it can possibly happen that a train occurs on a particular track it is stated as a safety property for all tracks that a collision has to be avoided. Eisner (in [Eis99]) used the fact that the states to be checked need not to be reachable by the system to reduce the model checking effort when applying the symbolic model checker SMV ([McM93]).

Using the FDR, we follow a different approach: Since we do not want to check situations in the system that can never occur we introduce trains into the system that *use* the specified Interlocking Model. For the behaviour of the trains, we assume that they behave in a 'reasonable' manner, e.g., they cannot jump but have to move from track to track. Once we have introduced this additional component into the Interlocking Model it also allows us to model the Signalling Principles in terms of trains. It can be seen that the Principle Model in terms of trains is much simpler. We do not have to derive the restrictions for tracks, routes, points, and signals such that safety is guaranteed but we are able to simply formalise a *safe movement of trains* on the track-layout. Safety properties specified in terms of other interlocking components can be found in [SWD97], however, it remains unclear how they are derived.

When modelling the behaviour of trains in the system we have to clarify the assumptions that we used. For instance, in the current model it is assumed that trains always stop at a red signal. We do not consider that the train-driver might fail to halt in time but might run into the next track, the overlap. We also simplify our notion of trains to those which are not longer than the length of two tracks.

4.1 The Interlocking Model

Firstly, we have to model the *static* information that is given by the track-layout. We identify the entities of the system accordingly to the given track-layout of MiniAlvey (see Figure 1), the layout we want to investigate. We define the tracks {`tac,tba,tad,tae,taz,tab`}, a number of signals {`s8,s12,s14`}, two points {`p201,p202`}, and the routes {`r12_1m,r14_1m,r8_1m,r8_2m`}, which are all main routes[2]. For any other layout, we can easily exchange the definition of these entities by reading them automatically from the given track-layout file. Moreover, we define two trains that are moving in the system, namely {`FS,CR`}. We also statically define a number of relations between these entities, which we derive from the given track-layout as well. For instance, each track has a number of *next tracks*, which are determined through the function `next()`. Usually each track has only one next-track. Only if a track contains a point, two different successor tracks are possible. The function `nextNormal(p201)`, for instance, specifies the next track if the point `p201` is set normal. All these functions are specific for a particular layout and have to be exchanged accordingly if we want to check a different track-layout. Other relations are derived from the Control Tables for the MiniAlvey. For instance, the tracks that need to be cleared before a route can be set, the entry-signal of each route, and the points that need to be locked and detected for each route.

Secondly, we specify the *dynamic* part of the system. This part is general for all track-layouts and must not be changed. Guided by the features of the CSP language, we model the behaviour of an interlocking system by means of several processes that interact with each other: Points, Signals, Timers, and Routes. The control of the system is decentralised, i.e., each process *actively* controls its own actions. In order to be able to check the safety of the system in terms of moving trains, we add a process that models trains that move according to the given situation in the interlocking system. To build up the overall system we put these processes in parallel in such a way that they constrain each other's behaviour: processes have to *synchronise* on the actions they may take. The movement of a train, for instance, is constrained by the current aspect of the signal in front of it; locking of a route depends on the current state of the points that are in the route, etc. We can draw a diagram for each operation that can be executed by the system (see Figure 2): each process may or may not constrain the other processes and may change its own data values on the channels.

The different process types as given in Figure 2 (Trains, Points, etc.) are modelled as sets of single processes of this type that are put in parallel. For instance, the process of Trains is defined as

```
Trains =
  [| {| SetRoute, ClearSignal,
       CancelRoute |} |]id:TrainId @ Train(id)
```

These sets of process of the same type have to synchronise over those actions that are guarded by a predicate that ranges over *all* entities of that type. For instance, *for all* trains it must be the case that they are not in the route in order to set this route (they have to synchronise on `SetRoute`) and furthermore all trains have to agree on the operations

[2]Note, that we omit in this model possible shunting and other routes

for clearing a signal (`ClearSignal`) and cancelling a route (`CancelRoute`).

Each single process (e.g., a train `Train(id)`, a point `Point(id)`, etc.) is given as a parameterised template, which is instantiated for every entity of the interlocking system (for trains $\texttt{id} \in \{\texttt{FS, CR}\}$). These templates model the rules for a suitable behaviour of an interlocking system according to its functional specification. In the following, we explain the scheme for modelling interacting processes using the example of trains and their movement, which is restricted by the processes `Points` and `Signals`. This should provide the reader with an understanding of our basic modelling approach.

The process `Train(id)` models the functionality of a train. An instantiation of the parameter `id` with a particular train provides a process-instance with its initial internal state. Generally, the "state" of a process is transferred via process parameters which are used as data values in the channels (which is an event attached with data). We model, for instance, the state of a train as the position of its front and rear. These are initially both set to the track `tac` for the train `CR` and both to the track `tba` for the train `FS` (see below the else case).

```
Train(id) = if id == CR
     then BehaveTrain(id.tac.tac)
     else BehaveTrain(id.tba.tba)
```

The sub-process `BehaveTrain` specifies the rules for the behaviour of the train. In the example above, `BehaveTrain(CR.tac.tac)` is the process that models the behaviour of train `CR` which is currently located on the track `tac` with its front and its rear. The parameter for front- and rear-position can be changed dynamically by the process. That is, the train can move. The process `BehaveTrain(id.ffront.rear)` is defined as a list of choices for the actions a train can take. We describe the first two cases as an example:

```
BehaveTrain(id.ffront.rear) =
    (([] t:next(ffront) @ ffront==rear &
        Moveff.ffront.t ->
         BehaveTrain(id.t.rear))
    []
    ( ffront!=rear &
        Mover.rear.ffront ->
         BehaveTrain(id.ffront.ffront))
    []...)
```

In our model, we distinguish whether the front and the rear of the train are on the same track or on different tracks. According to this difference we introduce two different kinds of movements: movement of the front (`Moveff`) and movement of the rear (`Mover`). With these two actions a train can change its state. Both events contain data (they are in fact channels, see above) that precise the current location and the next location. `Moveff.ffront.t` specifies a movement from the track where the front currently resides to some other track `t`. Since `t` cannot be any track but must be the next track in the track-layout, this value is chosen from the set of next tracks given by the function `next`; `[] t:next(ffront)` models this choice. The condition `ffront == rear` guards the movement of the front, i.e., only if both parts on the same track the front may move forward. Otherwise another action has to be chosen, e.g., the movement of the rear (the second branch of the choices). After an action is taken the process behaves like `BehaveTrain` but with changed parameters: the front is now on track `t`, the train has moved.

The movement of a train is also restricted through the state of signals, routes and points. In CSP, however, it is not possible to refer to parameters of other processes directly. We use instead the synchronisation mechanism which is introduced above: the signal-, point-, and route-processes have to agree and the events `Moveff` and `Mover` in the following way:

- If a train moves (it's front or rear) from a track where a point is located the next track is determined by the points direction.
- A train can only move from a track where a signal is located if the aspect of the signal is not red.

We build up the system gradually by putting processes in parallel. The process `Trains` models the set of all train-processes that are running in parallel. `Points` is the process of all points. Based on these two processes we can build up bigger processes, e.g., the process of trains and points `TrainPoints` as shown below. The first restriction for the movement of trains that we discussed above is treated when putting trains and points together in the following way:

```
TrainPoints =
 Trains [| {| Moveff.homeTrack(p201),
         Moveff.homeTrack(p202),
         Mover.homeTrack(p201),
         Mover.homeTrack(p202) |} |] Points
```

Based on the track-layout, `homeTrack()` is a static function that returns the location of a point. Since the movement of a train is restricted through points only if the train is on a track where a point is located, the processes have to synchronise only over movements from particular tracks, namely `homeTrack(p201)` and `homeTrack(p202)`. The second restriction for movements of trains is treated in a similar way. Based on the combined process TrainPoints and the process of all signals, `Signals` , we combine a larger process, called `TrainPointSignals`, in which trains, points and signals interact:

```
TrainPointSignals =
 TrainPoints [| {| Moveff.homeSignal(s8),
         Moveff.homeSignal(s12),
         Moveff.homeSignal(s14) |} |] Signals
```

The static function `homeSignal()` returns the track where a signal is located. The restriction through signals is only important for the movement of the front of a train. The movement of the rear (modelled by action `Mover`) is not affected by this restriction.

The general functionality of an interlocking system is specified in our interlocking model by means of 11 operations. Each of these operations can be presented as diagrams of interacting processes as shown in Figure 2. This includes moving of trains, replacing signals (i.e., setting the aspect to red), freeing the locking of a point, setting a route, clearing a signal (i.e., setting its aspect to yellow), locking of a route when a train is approaching, moving a point (which is completed with a time delay), releasing of a route in two steps (firstly a train must be recognised on the locked route, secondly the train is leaving the route or if a train is halting within the route for some time), cancelling of the route (with freeing or not freeing the points in the route), pulling a button in order to manually set a signal aspect to red.

All these operations are actively processed by one or more components and also guarded by the local state of some components of the interlocking model. This kind of conditional operations is modelled via synchronisation as shown above. The interested reader may find an abstract description of these operations as well as the full CSP model our technical report [RTW01].

4.2 The Principle Model

The two basic safety properties we want to prove for our layout are the following:

- *No collision*: It never happens that two trains in the system collide. That is, not two trains are running on the same track at the same time.
- *No derailment*: No derailment of a train can occur. That is, when a train is passing a track with a point, this point will not be moved. (We do not currently consider other causes for derailment.)

These two safety properties are standard requirements for an interlocking system and are based on a simplification of our work on signalling principles. We formalise these principles in terms of trains. This provides a much higher-level of the description than a formalisation in terms of routes, points and signals and will ease the modelling task. Once they are formalised appropriately, they can be applied to each track layout. We model these general principles as two CSP processes. The FDR tool then allows us to check the refinement between the principle model and the MiniAlvey model. The general idea of modelling a principle is that we model a *good* behaviour that does satisfy the principle. That is, we take a model of arbitrary behaviour for any possible action and restrict this model such that it behaves for particular actions in a prescribed, good way. In the following we describe more concretely how this works for our principles.

No collision of trains. In our simplified model, two trains do not collide if they never move to the same track. Moving onto the same track can only happen when a train moves with its front onto an occupied track. We model this situation by means of three processes, `P`, `SafeMove`, and `CHAOS`, in the following way: We design a process that can store the information of two sets of tracks: one for the tracks that are occupied by the front and one set for tracks that are occupied by the rear of a train. Whenever a train moves with its front onto a next track, it has to check if this next-track is not already occupied, i.e., if this track is already in one of the sets of occupied tracks. If the next track is not currently occupied and the train moves onto this track, the sets of tracks that are occupied by a train-front is updated correspondingly (i.e., the old track is taken out of this set and the next track is added). This can be modelled in CSP by following process `P`:

```
P(F,R) = [] on:union(F,R) @
[] nextTr:next(on) @
   ( not(member(nextTr,union(F,R))) &
      Moveff.on.nextTr ->
       P(union(diff(F,on),nextTr),R)
   []
      Mover.on.nextTr ->
       P(F,union(diff(R,on),nextTr))
   []
      STOP
   )
```

The parameters of P are the sets of tracks. Parameter `F` is the set of tracks occupied by a front, `R` is the set of tracks occupied by a rear. The process has three choices of actions: either try to do a move with the front (`Moveff.on.nextTr`) or do a move with the rear (`Mover.on.nextTr`) or do nothing (`STOP`). Whenever the process chooses an action for moving, the parameters of the process (i.e., the sets of occupied tracks) are updated correspondingly. Given this process `P`, we are now able to model the good behaviour of trains as the process

```
SafeMove = P({tba,tac},{tba,tac})
```

where the parameters of `P` are initialised with the tracks that are occupied by train-fronts (first parameter) or train-rears (second parameter) in the initial state (i.e., train `CR` is initially on track `tac`, and train `FS` is initially on track `tba`). `SafeMove` is now put in parallel with a process called `CHAOS` which is a built-in process of CSP. This process behaves arbitrarily over a given set of possible actions. In our case these actions are all actions of the interlocking system (`Events`) except the move actions, `Moveff` and `Mover` (the operator `diff` computes the set difference). The move actions are excluded because the movement of trains should happen in an ordered way, namely like it is specified in `SafeMove`.

The expression

```
NoCollisionNetwork =
      SafeMove ||| CHAOS(diff(Events,
                          {|Moveff, Mover|}))
```

defines the principles process for no collision in terms of `SafeMove` and the `CHAOS` process. The latter process does every action apart from moving which is organised according to `SafeMove`. Given this, we can now check that the process `NoCollisionNetwork` is refined by the process of the interlocking system `Network`.

No derailment of trains. In our simplified model, a derailment of a train can be caused by moving a point when a train is moving over its track. This is modelled by a process `NoDerail` which is similar to the process `SafeMove` described above:

```
NoDerail = Q({tba,tac},{tba,tac})
```

where `Q` is a process that models the good behaviour. `Q`, as the process `P` above, has two parameter sets that store the sets occupied tracks. Whenever a train moves from a track onto a new track, these parameter sets are changed correspondingly. `Q` also prescribes the action `MovePt` in such a way that moving a point is only possible if its home-track is not currently occupied by a train (i.e., this track is not member of one of the parameter-sets). `Q` is modelled as follows:

```
Q(F,R) =
   ([] t:TrackId @ [] p:PointId @
    [] d:Direction @
    not(member(t,union(F,R)))
      and t==homeTrack(p) &
     MovePt.p.d -> Q(F,R)
   )
 []
   ([] on:union(F,R) @ [] nextTr:next(on) @
      ( Moveff.on.nextTr
          -> Q(union(diff(F,on),nextTr),R)
      []
        Mover.on.nextTr
          -> Q(F,union(diff(R,on),nextTr)))
      ))
 []
    STOP
```

We put the process `NoDerail` in parallel with a process `CHAOS` which behaves arbitrarily over all possible actions except the actions `Moveff`, `Mover`, and `MovePt`. The behaviour on these actions is prescribed by the process `NoDerail`.

```
NoDerailmentNetwork =
 NoDerail ||| CHAOS(diff(Events,
                    {| Moveff,Mover,MovePt}))
```

The resulting process NoDerailmentNetwork is now checked again the process Network, that models the interlocking system. If a counterexample can be found then this will indicate a violation of the modelled good behaviour of the process NoDerail. That is, a derailment could possibly happen.

5 Checking Results

In order to give an impression of the counter-examples that are provided by the FDR tool in the case that a violation of the principles could be found, we show four examples from the many we obtained during our work with the tool. A counter-example is a possible run of the model that leads to a state in which the checked principle is not satisfied. All four counter-examples that are listed below violate the principle of no collision (in some counter-examples the reader might also find a violation of the no derailment requirement). The format of this counter-example is a sequence of actions that the processes in the model (Trains, Points, Signals, Routes) have agreed on.

1. A route could be set to route-reverse (SetRoute) although some points in the route were not locked and also leading into the wrong direction. The FDR tool outputs the following counter-example:

```
SetRoute.r12_1m
ClearSignal.s12.r12_1m
Aq_Applock.s12.CR.r12_1m
Moveff.tac.tad
Replace_Signal.s12
Mover.tac.tad
Pull_Button.s12
Moveff.tad.tae
Mover.tad.tae
Section_Release_Applock.r12_1m.CR
CancelRouteLock.r12_1m
FreePoint.p202.rtN
SetRoute.r8_2m
ClearSignal.s8.r8_2m
Aq_Applock.s8.CR.r8_2m
Moveff.tae.taz
Mover.tae.taz
Moveff.taz.tab
Mover.taz.tab
MovePt.p201.reverse
Moveff.tab.tba
```

   This sequence of actions indicates the following erroneous behaviour: The train CR can move along route r12_1m, which was set before. It continues to move to tracks tae, taz, and tab after setting route r8_2m (SetRoute.r8_2m). Although the route is set, the point p201 is able to move, i.e., it obviously is not locked (since this is a condition for moving a point). This causes the collision of the two trains on track tba when the train CR is moving on the wrong route. Figure 3 illustrates the movement that leads to the collision of the trains. The thick arrow shows which way train CR moves.

   Analysing our model, we found that only one point in the route r8_2m is locked when the route is set, namely p202. The other point, p201, did not contribute to the action SetRoute.r8_2m. We changed the model such that all points in a route (this includes also the points in the overlap) have to agree on this action, i.e., all points in the route have to be set properly and have to be locked.

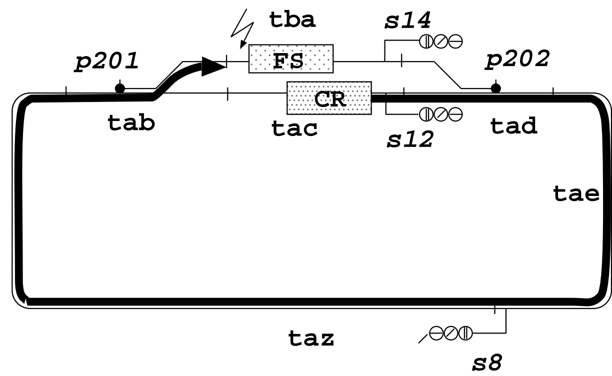

Figure 3: Collision of the two trains, example 1

2. Acquiring an approach-locking did not necessarily happen after a signal was cleared. As a consequence, the corresponding route was not immediately approach-locked and the locking of points in the route could be freed again and the points could move after the signal was cleared. Two colliding routes could be set at the same time. We changed the model such that we specified the actions ClearSignal and Aq_Applock as a sequence: Whenever ClearSignal is executed the next action of the signal must be Aq_Applock.

3. We may also, on purpose, introduce a bug into the model in order to see if the given counter-example is reasonable and understandable. For this test, we delete one of the tracks in the static definition of routes. Originally, the route r8m_2m consists of the tracks {taz, tab, tac, tad}. We delete the track tac from this set and as a consequence get the following counter-example when running the checks for a possible collision of trains:

```
MovePt.p202.reverse
CompleteMove.p202
SetRoute.r14_1m
ClearSignal.s14.r14_1m
Aq_Applock.s14.FS.r14_1m
Moveff.tba.tad
Replace_Signal.s14
Mover.tba.tad
Moveff.tad.tae
Mover.tad.tae
FreePoint.p202.rtR
Pull_Button.s14
Section_Release_Applock.r14_1m.FS
CancelRouteLock.r14_1m
MovePt.p202.normal
CompleteMove.p202
SetRoute.r8_2m
ClearSignal.s8.r8_2m
Aq_Applock.s8.FS.r8_2m
Moveff.tae.taz
Replace_Signal.s8
Mover.tae.taz
Moveff.taz.tab
Mover.taz.tab
Moveff.tab.tac
```

   The sequence of actions shows that in the end the moving train runs into a train that is still in its initial position on track tac. This track, since it is deleted from the tracks of route r8_2m, was not be checked when freeing the route for being used. Figure 4 sketches the movement that leads to the collision.

The counter-examples output by FDR are very useful for debugging our formalisation of the inter-

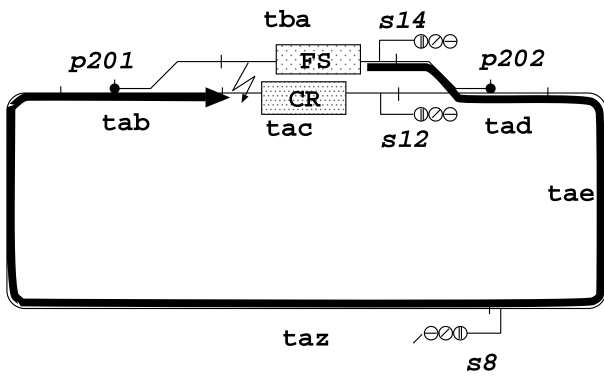

Figure 4: Collision of the two trains, example 3

locking system. It is worth noting that even without knowledge of the formal interlocking model the practitioners were able to understand the generated cases just by reading the meaningful names of the occurring events in the sequence.

6 Discussion

Model checking has been applied before to the analysis of interlocking systems: Gnesi et. all ([GLL+00]), Bernardeschi et. all ([BFGM96]), and Cleaveland et. all ([CLN96]), for instance, have addressed the problem of fault-tolerance in interlocking systems. In their work, the checking task is focussed on communication issues between components of the system rather than the control logic of the interlocking. The preferred modelling language for formalising the system are based on process algebras (e.g., CSP, CCS, PROMELA). These languages provide suitable features for modelling communication between components.

Closer to our approach, is the work of Simpson, Woodcock and Davies ([SWD97]). The formal notation CSP is used for modelling the control logic of an interlocking system and the FDR model checker is applied to check the safety properties. However, their model is at a lower level of abstraction than ours. The safety invariants, namely no collision of trains and no derailment, are modelled in terms of the system's components such as points, signals, routes, etc. This formalisation of safety invariants has to be manually derived from the track-layout (in the paper it is not explained how) and, therefore, it is not obvious if a given set of invariants is complete and covers all eventualities.

Eisner ([Eis99]) used the symbolic model checker SMV to analyse the interlocking logic of a given railway yard. This approach benefits from the fact that the Signalling Principles (i.e., the safety requirements) are generally stated for all states of the system. This includes also states that are practically not reachable (e.g., non-adjacent tracks are occupied by one train). Due to the internal algorithms for symbolic model checking and for the particular form of safety requirements that is often used, the checking effort turned out to be more efficient if all states are covered - not only the reachable ones.

We approached this observation differently: Since FDR is not a symbolic model checker our approach does not benefit from also checking the non-reachable states. Therefore, we restricted our model to possible behaviour, i.e., the reachable states, by introducing the notion of *trains* into the model. That is, having a formalisation that models the interlocking system, we model trains that *use* this system. They are allowed to move arbitrarily along the tracks of the given layout but have to follow the "general laws of train-driving", for instance, trains have to stop at red signals and they are not supposed to jump or move backwards. These "laws" can be seen as *assumptions* that we made on train movements. A careful analysis of the appropriateness of our assumptions needs to be made by the practitioners. An ongoing discussion with experts from QR targets this issue.

Introducing trains not only helps to reduce the number of states that the model checker has to investigate but also lightens the task of modelling the Principle Model, i.e., the safety requirements for the interlocking system. As given in Section 4.2, the principles were fairly easy to model by means of trains running on the system.

Using CSP as a modelling language and FDR as the corresponding model checker is our first attempt for supporting the analysis of interlocking systems in the early stages of design. However, we found that modelling languages based on process algebras, such as CSP, are not very well suited for describing the content of a Control Table. The CSP-based models of the interlocking system and the signalling principle is difficult to understand and validate by the practitioners and thus does not yield a good documentation for QR. However, we found that the counter-examples that were output by the FDR tool were easily read by railway experts (even without knowledge of CSP). In our future work, we are pursuing a different approach that uses a kind of guarded command language that can be model checked. We wish to ease the understanding of the formal model if it is closer related to the tabular form that practitioners use. The ASM/SMV approach suggested by [CW00] might be suitable: the language of Abstract State Machines (ASM) can be used to model the conditioned operations of the interlocking system and an interface to the model checker SMV provides tool support for checking.

7 Acknowledgements

The work described in this paper is a result of a joint project with Queensland Rail (QR). I would like to thank George Nikandros and David Barney from QR for their interest in our work and their helpful comments on this paper. I also want to thank Neil Robinson and David Tombs for helping to improve draft versions of this paper and Graeme Smith for the inspiring discussions on CSP and the use of FDR. Thanks also to the reviewers for their useful comments on a draft of this paper.

References

[BFGM96] C. Bernardeschi, A. Fantechi, S. Gnesi, and G. Mongardi. Proving safety properties for embedded control systems. In *Procs. of Conference on Dependable Computing (EDCC-2)*, volume xvi+440, pages 321–332. Springer-Verlag, 1996.

[CLN96] R. Cleaveland, G. Luettgen, and V. Natarajan. Modeling and verifying distributed systems using prioriteis: A case study. In *Procs. of Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 287–297. Springer-Verlag, 1996.

[CW00] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. of 6th Int. Conference for Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, volume 1785 of *LNCS*, pages 331–346. Springer-Verlag, 2000.

[Eis99] C. Eisner. Using symbolic model checking to verify the railway stations of hoorn-kersenboogerd and heerhugowaard. In *Proc. of the 10th IFIP Working*

Conference on Correct Hardware Design and Verification Methods (CHARME'99), volume 1703 of *LNCS*, pages 97–109. Springer-Verlag, 1999.

[GLL+00] S. Gnesi, G. Lenzini, D. Latella, C. Abbaneo, A. Amendola, and P. Marmo. An automatic spin validation of a safety critical railway control system. In *Procs. of IEEE Conference on Dependable Systems and Networks*, pages 119–124. IEEE Computer Society Press, 2000.

[McM93] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[RBK+01] N. Robinson, D. Barney, P. Kearney, G. Nikandros, and D. Tombs. Automatic generation and verification of design specification. In *Proc. of Int. Symp. of the International Council On Systems Engineering (INCOSE)*, 2001.

[Ros94] A. W. Roscoe. Model-Checking CSP. In *A classical Mind, Essays in Honour of C.A.R. Hoare.* Prentice Hall, 1994.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[RTW01] N. Robinson, D. Tombs, and K. Winter. Signalling design tools - requirements feasibility report. Technical report, SVRC, The University of Queensland, 2001.

[SWD97] A. Simpson, J. Woodcock, and J. Davies. The mechanical verification of solid state interlocking geographic data. In *Proc. of Formal Methods Pacific (FMP'97)*, volume vii+320, pages 223–243. Springer-Verlag, 1997.