Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

This is a pre-print of an article published in STTT 20(3)
The final read-only version is available from Springer Nature's at: http://rdcu.be/lD78
The final authenticated version is available online at: http://dx.doi.org/10.1007/s10009-018-0488-3

Towards Formal Methods Diversity in Railways: an Experience with Six Formal Tools

Franco Mazzanti, Alessio Ferrari and Giorgio O. Spagnolo

ISTI-CNR, Via G. Moruzzi 1, Pisa, ITALY,
e-mail: {firstname}.{lastname}@isti.cnr.it,
WWW home page: http://fmt.isti.cnr.it

Received: date / Revised version: date

Abstract. Several formal tools exist that can be exploited to validate early system designs, and that are applicable to problems of the railway domain. In this paper, we present an experience in formal modelling and verification using six different formal environments, namely UMC, Promela/SPIN, NuSMV, mCRL2, CPN Tools and FDR4. In particular, we model and verify an algorithm that addresses a typical railway problem, namely deadlock avoidance in train scheduling. The algorithm is designed according to a prototypical architecture, the so-called *blackboard* pattern, in which a a set of global data is concurrently and atomically updated by a set of concurrent guarded agents. Our experience shows that the design of the algorithm can be translated into the different formalisms with limited effort, while deep proficiency with the tools is required to optimise the performance. The current paper establishes the preliminary foundations for the concept of *formal methods diversity* in the development of railway systems. The concept is based on the idea that, if different non-certified formal environments are used to verify the same design, this increases the confidence on the verification results. The industrial application of this promising concept requires further research, and appropriate guidelines shall be established to identify the proper formal environments to use for a specific railway problem, and to define an industrial process in which formal methods diversity can be exploited at its full benefits. The paper presents the different models developed, compares the tools employed in terms of ease-of-use and performance, and discusses the industrial implications of the concept of formal methods diversity in the railway domain.

Key words: Formal Methods Diversity, Model Checking, Deadlock Avoidance, Train Scheduling, Railways, Automatic Train Protection, CBTC.

1 Introduction

The CENELEC EN 50128 norm [8], for the development of railway safety-critical software, recommends the usage of formal methods during the design and implementation of railway products. Several industrial experiences have been documented in the literature concerning the formal development of railway software [18,32, 60]. The usage of the B method [2] for the development of the SACEM system, a control platform for a line of Paris RER [13], and the iterative formal verification of the Paris automatic metro line 14, also based on the B method [5], are successful, early experiences that have shown the practicability and effectiveness of formal methods to railway companies. With the advent of model checking techniques and tools [12], experiences on the application of these approaches were performed in railways, especially for what concerns the validation of interlocking systems [59,58,37,23,7,42]. More recently, formal model-based approaches, involving graphical modelling and code generation, were also used for the development and verification of railway systems, with a main focus on automatic train control (ATC) and protection (ATP) systems [22,40,54,10,20]. Some experiences were also performed on the usage of Coloured Petri Nets (CPN) for modelling and simulation of railway signalling systems [57,43].

Although formal tools exist that are certified according to the EN 50128 norm, as, e.g., SCADE [16] from Esterel Technologies, the majority of the formal environments available are not certified. Hence, notwithstanding the usefulness of formal methods for discovering design flaws early in the development, the result of a for-

Send offprint requests to:

mal modelling and verification process in which a non-certified tool is used cannot be considered as a final proof of the correctness of a certain design with respect to the verified properties. On the other hand, the existence of different, non-validated, tools producing the same results might increase the overall confidence of the verification outcomes. This principle was previously applied in the avionic domain by Rockwell Collins [51], which, in collaboration with other partners, developed translators from semi-formal models expressed in Simulink/Stateflow towards the Lustre formal language [34], and then towards formal environments, such as PVS [52] and NuSMV [11], in which design properties and system requirements can be verified. However, to our knowledge, no equivalent experience exists in the railway domain. We hypothesise that this might be due to the perceived difficulty of formal methods for railway practitioners, and to the common idea that, if mastering a single formal tool is a problem, mastering more than one might be hardly feasible.

In this paper, we show that a representative railway problem can be modelled and verified with limited effort using six different tools, namely: UMC [55], Promela/SPIN [39], NuSMV [11], mCRL2 [31], FDR4 [28] and Coloured Petri Nets (CPN) Tools [44]. In particular, we modelled an algorithm for deadlock avoidance in train scheduling. The algorithm was previously implemented as part of an Automatic Train Supervision (ATS) system [49,50] of a Communications-based Train Control System (CBTC) [24]. Such system controls the movements of driverless trains inside a given yard. The deadlock avoidance algorithm takes care of avoiding situations in which a train cannot move because its route is blocked by another train. Equipped with this algorithm, the ATS is able to dispatch the trains without ever causing situations of deadlock, even in presence of arbitrary delays with respect to the planned timetable. This kind of problem is a rather typical one – not only for the railway domain [15] – which can be modelled as a set of global data that is concurrently and atomically updated by a set of concurrent guarded agents – i.e., agents that, when certain global conditions are met, are allowed to atomically change the global status. This design strategy is normally referred as the *blackboard* architectural pattern [15]. In this paper, we show the design of the algorithm, the different models produced with the six formal tools, and the results of the verification activities, observing differences and hurdles in the usage of the six environments.

This paper establishes a preliminary basis for the potential usage of formal methods *diversity* in the design and verification of railway software. In particular, our experience tells that, given a system design, limited effort and adjustments are required to translate the design into different formalisms. However, from our experience, we saw that small choices in the specification of the models, or in the verification options, can greatly impact on the verification time. With some differences, this observation holds for all the modelling frameworks considered. Hence, we argue that a deep proficiency with each one of the frameworks is required to effectively exploit their verification capabilities.

The paper extends a previous contribution to the ISoLA 2016 conference [48]. With respect to this previous work, the current one describes the experience with two additional environments, namely CPN and FDR4 (Sect. 8 and 7), provides a more in-depth discussion and comparison among the six tools (Sect. 9), and discusses the potential of formal methods diversity in the railway domain (Sect. 10).

The rest of the paper is structured as follows. In Sect. 2 we describe the deadlock avoidance algorithm that we modelled. In Sect. 3–7, we show our models and the verification results for UMC, NuSMV, Promela/SPIN, mCRL2, FDR4 and CPN Tools, respectively[1], and, within the descriptions of the models, we highlight the peculiarities of the different languages and environments. Finally, Sect. 11 concludes the paper and provides general observations on the experience.

2 The Deadlock Avoidance Algorithm

This section describes basic elements of the modelled algorithm, which was defined in our previous works [49, 50]. Fig. 1 shows the structure of the railway layout considered in this study. Nodes in the yard correspond to itinerary endpoints, and the connecting lines correspond to the entry/exit itineraries to/from those endpoints. Eight trains are placed in the layout. Each train has its own mission to execute, defined as a sequence of itinerary endpoints. For example, the mission of `train0`, which traverses the layout from left to right along top side of the yard, is defined by the mission vector: $T_0 = [1, 9, 10, 13, 15, 20, 23]$. The mission of `train7`, which instead traverses the layout from right to left, is defined by the vector: $T_7 = [26, 22, 17, 18, 12, 27, 8]$. The progress status of each train is represented by the index, in the mission vector, which allows to identify the endpoint in which the train is at a certain moment. We will have 8 variables $P_0, \ldots, P_7$, one for each train, which store the current index for the train. For example, at the beginning, we have $P_0 = 0, \ldots, P_7 = 0$, since all the trains occupy the initial endpoints of their missions – at index 0 in the vector.

If the 8 trains are allowed to move freely, i.e., if their next endpoint is free, there is the possibility of creating deadlocks, i.e., a situation in which the 8 trains block

[1] All the verification experiments have been conducted on a Mac Pro (late 2013) workstation with *Quad-core 3,7Ghz Intel Xeon E5, 64 GB RAM* running OS X 10.11 (El Capitan).
All the models referred in this paper can be retrieved from the URL `http://fmt.isti.cnr.it/WEBPAPER/STTT2017data.zip`

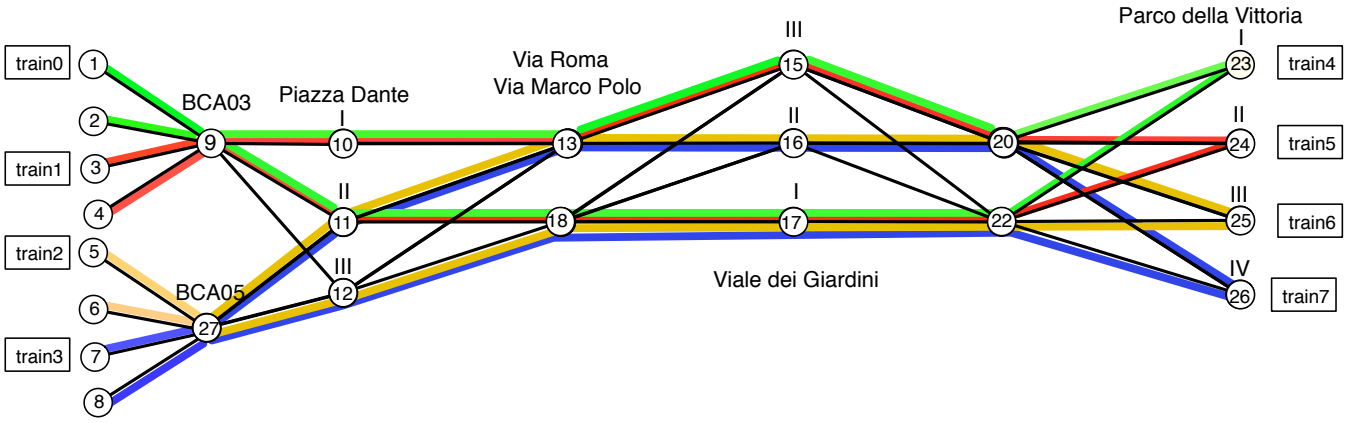

Fig. 1: A fragment of the yard layout and the 8 missions of the trains

each other in their expected progression. To solve this problem the scheduling algorithm of the ATS must take into consideration two *critical sections* A and B – i.e., zones of the layout in which a deadlock might occur, – which have the form of a ring of length 8 (see Fig. 2), and guarantee that these rings are never saturated with 8 trains – further information on how critical sections are identified can be found in our previous work [49,50]. This can be modelled by using two global counters RA and RB, which record the current number of trains inside these critical sections, and by updating them whenever a train enters or exits these sections. For this purpose, each train mission T_i, with $i = 0 \dots 7$, is associated with: a vector of increments/decrements A_i to be applied to counter RA at each step of progression; a vector B_i of increments/decrements to be applied to counter RB.

For example, given $T_0 = [1, 9, 10, 13, 15, 20, 23]$, and $A_0 = [0, 0, 0, 1, 0, -1, 0]$, when `train0` moves from endpoint 10 to endpoint 13 ($P_0 = 3$) we must check that the $+1$ increment of RA does not saturate the critical section A, i.e., $RA + A_0[P_0] \leq 7$; if the check passes then the train can proceed and safely update the counter $RA := RA + A_0[P_0]$. The maximum number of trains allowed in each critical section (i.e., 7), will be expressed as LA and LB in the rest of the paper.

The models presented in the following sections, which implement the algorithm described above, are deadlock-free, since the verification is being carried on as a final validation of a correct design. The actual possibility of having deadlocks, if the critical sections management were not supported or incorrectly implemented, can easily be observed by raising from 7 to 8 the values of the variables LA and LB.

The current design, in which each train movement logically corresponds to an atomic system evolution step, leads to a state-space of 1,636,537 configurations.

3 The UMC Model

UMC [55] is a model checker that belongs to the KandISTI[2] [56] family. Its development started at ISTI in 2003 and has been since then used in several research projects. So far UMC is not really an industrial scale project but more an (open source) experimental research framework. It is actively maintained and is publicly usable through its web interface[3].

The KandISTI family comprises four model checkers, each of which is oriented to a particular system design approach, but all of which share the same underlying abstract model and verification engine. The basic underlying idea behind KandISTI is that the evolution in time of the system behaviour can be seen as a graph where both edges and states are associated with sets of (composite) labels [30]. Labels on the states represent basic state properties, and labels on the edges represent properties of system transitions. The logic supported by the KandISTI framework uses the evolution graph as semantic model and allows to specify abstract properties in a way that is rather independent from the internal implementation details of the system [19].

The different flavours of the various KandISTI tools have to do with the choice of one of the supported specifications languages, which range from process algebras to sets of UML-like statecharts. In our case, we will use the UMC tool, since we considered it the most adequate to model our algorithm. In UMC, a system is described as a set of communicating UML-like state machines. In our particular case, the system is composed of a unique state machine, in which we have a `Vars` part – including the global state – and a `Behavior` part – specifying the state machine behavior.

The `Vars` part contains the vectors describing the train missions (T_i), the indexes recording the train progresses (P_i) – i.e., the indexes in the previous vectors –, the occupancy counters RA and RB of the two crit-

[2] `http://fmt.isti.cnr.it/kandisti`
[3] `http://fmt.isti.cnr.it/umc`

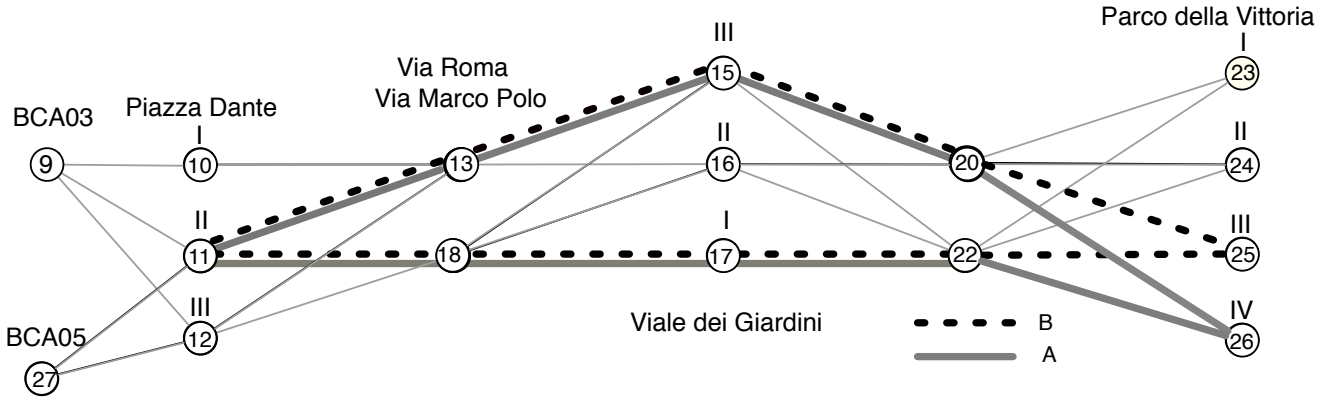

Fig. 2: The critical section A and B which must not be saturated by 8 trains

ical sections, and the vectors A_i, B_i including the increments/decrements that should be performed by the trains at each step of their progress for the critical sections A and B, respectively. In addition, we have the two constants indicating the maximum number of trains allowed in the critical sections (LA, LB).

```
Vars:
  --  mission steps for train0
  T0: int[] := [ 1, 9,10,13,15,20,23];
    . . .
  --  mission steps for train7
  T7: int[] := [26,22,17,18,12,27, 8];
  -- limit value for region RA
  LA: int :=7;
  -- RA updates steps for train0
  A0: int[] := [0, 0, 0, 1, 0,-1, 0];
    . . .
  -- RA updates steps for train7
  A7: int[] := [1, 0, 0, 0,-1, 0, 0];
   -- occupancy of region RA
  RA: int :=1;
   -- limit value for region RB
  LB: int :=7;
  -- RB updates steps for train0
  B0: int[] := [0, 0, 0, 1, 0,-1, 0];
    . . .
  -- RB updates steps for train7
  B7: int[] := [1, 0, 0, 0,-1, 0, 0];
  -- occupancy of region RB
  RB: int :=1;
  -- train progresses
  P0,P1,P2,...,P7:int :=0;
```

In this particular case the size of a state is fixed and static. However, this is not a requirement for UMC, since we can have variables representing unbounded vectors, queues, unbounded integers, which together with the (potentially unbounded) events queues can contribute to make the actual size of a state highly dynamic. This dynamism might lead to potentially infinite state systems.

In the `Behavior` part of our class definition we will have one transition rule for each train, which describes the conditions and the effects of the advancement of the train. A generic transition rule is expressed as follows:

```
<SourceState> -> <TargetState>{
                 <EventTrigger>[<Guard>]/<Actions>
                 }
```

A transition rule expressed as above intuitively states that when the system is in the state *SourceState*, the specified *EventTrigger* is available, and all the *Guards* are satisfied, then all the *Actions* of the transition are executed and the system state passes from *SourceState* to *TargetState*.

The interleaving of the progress of the various trains is therefore modelled by the internal non-determinism of the possible applications of state machine transitions. In our case there is no external event that triggers the system transitions, therefore the transitions will be controlled only by their guards.

In the case of `train0`, for example, we will have the transition rule:

```
s1 -> s1
 {- [-- train0 has not yet completed its mission
     P0 <6 &
     -- next position not occupied by train1
     T0[P0+1] != T1[P0] &
      . . .    -- next position not occupied by ...
      -- next position not occupied by train7
     T0[P0+1] != T7[P7] &
      -- A is not saturated by arrival of train0
     RA + A0[P0+1] <= LA &
      -- B is not saturated by arrival of train0
     RB + B0[P0+1] <= LB
    ] /
       -- update occupancy of critical section A
      RA = RA + A0[P0+1];
       -- update occupancy of critical section B
      RB = RB + B0[P0+1];
       -- update train progress
      P0 := P0 +1;
 }
```

As a last step we have to define what we want to see on the abstract L2TS associated to the system evolutions. Indeed, the overall behaviour of a system is formalised as an abstract doubly labelled transition system (L2TS), and abstraction rules allow to define what we want to see as labels of the states and edges of the L2TS.

The abstraction rules are expressed in the `Abstraction` part of the specification, in which we define which labels should appear on the edges and states of the abstract evolution graph. In our case, we are interested to observe the existence of a certain state where all trains have completed all their missions. This can be done assigning a state label, e.g. `ARRIVED`, to all the system configurations in which each train is in its final position.

```
Abstractions  {
State SYS.P0=6 and
      SYS.P1=6 and
          . . .
      SYS.P7=6 -> ARRIVED
                -- abstract label on final node
}
```

At this point, the L2TS associated to our model will be a directed graph that will converge to a final state labelled ARRIVED in the case that no deadlock occurs in the system. The branching-time, state/event based temporal logic supported by UMC has the power of full μ-calculus but also supports the more high level operators of Computation Tree Logic (CTL). The property that for all executions all the trains eventually reach their destinations be easily checked by verifying the CTL-like formula:

```
AF ARRIVED
```

The **AF** operator inside the above CTL formula specifies that for all execution paths (**A**) of the system, eventually in the future (**F**), we should reach a state in which the state predicate `ARRIVED` holds.

If this property does not hold, UMC allows to interactively explore the set of system evolution steps that led to a failure (which in this case do have the shape of a single path but which in general may have the shape of a graph), and view all the internal details of the traversed states. One of the design goals of UMC is indeed the one of helping the user to easily understand the defects in its early designs, by exploiting an interactive explanation of the obtained evaluation results – not just a state-space fragment acting as counter-example.

In our case the formula is *true* and UMC completes the evaluation in a time which ranges from 28 seconds to 106 seconds depending on how the tool is used. The fastest results of 28 seconds is obtained by exploiting a prototypal parallel version of UMC [47], by adopting a depth-first exploration strategy, and letting the evaluation to proceed in a non-interactive way which does not collect the data necessary for a subsequent explanation of the results.

As an alternative modelling approach, we might have modelled the successful completion of all the train missions as an observable *event* on the graph. To achieve this we should introduce an additional evolution to the state machine, which generates the `Arrived` signal after all trains have completed their missions.

```
s1 -> s2 {- [P0=6 & P1=6 & ... & P7=6] / Arrived}
```

Furthermore, in this case we should associate an observable label in the abstract evolution graph, corresponding to the internal event of signal generation.

```
Abstractions  {
Action :  Arrived -> arrived
                -- abstract label on final edge
}
```

At this point the property to be verified becomes:

```
AF {arrived} true
```

The **AF** { } operator inside the above CTL formula specifies that for all execution paths (**A**) of the system, eventually in the future (**F**), we should reach a transition whose labels satisfy the action predicate `arrived`, and whose target state satisfies the formula `true`.

4 The NuSMV Model

NuSMV[4] [11] is a software tool for the formal verification of finite state systems. NuSMV was jointly developed by FBK-IRST and by Carnegie Mellon University. NuSMV allows to check finite state systems against specifications in the Computation Tree Logic (CTL), Linear Temporal Logic (LTL) and in the Property Specification Language (PSL)[1].

Since NuSMV is intended to describe finite state machines, the only data types in the language are finite ones, i.e. boolean, scalar, bit vectors and fixed arrays of basic data types. A state of the system is represented by a set of variables. Assignment rules in the language allow to specify *total* functions, which define all the possible values that a state variable can assume in the next state.

NuSMV distinguishes between system constants (`DEFINE` construct), and variables (`VAR` construct). The system constants are represented by the T_i, A_i, B_i and LA, LB data values:

```
DEFINE
   T0 := [ 1, 9,10,13,15,20,23];
    . . .
   T7 := [26,22,17,18,12,27, 8];
   LA := 7;
   A0 := [0, 0, 0, 1, 0,-1, 0];
    . . .
   A7 := [0, 1, 0, 0,-1, 0, 0];
   LB := 7;
   B0 := [0, 0, 0, 1, 0,-1, 0];
    . . .
   B7 := [1, 0, 0, 0,-1, 0, 0];
```

The state variables consist of the different P_i of the various train progresses, and of the occupancy status of RA and RB of the two critical sections. Furthermore, we will need an additional `RUNNING` state variable for modelling the non-determinism in the choice of the potentially moving train and consistently synchronise the updates of the P_i, RA, and RB variables.

[4] http://nusmv.fbk.eu/

```
VAR
   RUNNING: 0..7;
   P0:  0..6;
    . . .
   P7:  0..6;
   RA:  0..8;
   RB:  0..8;
```

The initial state, and the state transitions specifying the behaviour are expressed under the `ASSIGN` construct of a NuSMV mudule. The definition of the initial state is specified making use of the `init` operator:

```
ASSIGN
  init(P0) := 0;
    . . .
  init(P7) := 0;
  init(RA) := 1;
  init(RB) := 1;
```

The evolutions corresponding to the train movements, i.e., the system transitions, are specified making use of the `next` operator. For example, the evolution of `train0` is now described by the following rule:

```
next(P0) :=
   case
      -- train0 selected for possible movement
      RUNNING =0 &
      -- train0 has not yet completed its mission
      P0 < 6 &
      T0[P0+1] !=  T1[P1] &
      . . .  --  next place not occupied by other trains
      T0[P0+1] !=  T7[P7] &
      -- critical section constraints satisfied
      RA + A0[P0+1] <= LA &
      RB + B0[P0+1] <= LB &
      RA + A0[P0+1] >= 0 &
      RB + B0[P0+1] >= 0
            --  train0 advances one step
      :    P0+1;
      TRUE
            --  train0 not selected/not allowed to move
      :    P0;
   esac;
```

We must observe that the definition of the next value for the P_0 variable is now total. If the train can move, the value of P_0 is incremented, while if the train is not allowed to move, the value of P_0 in the next state remains the same. Notice that in this way we are introducing loops, in each node of the graph, corresponding to the dummy evolutions of trains which cannot actually move.

The definition of the next values of the RA variable should take into consideration again which train is selected for possible movements, and whether or not the train is actually allowed to move. Therefore, the transition definition for the RA variable now becomes:

```
next(RA):=
      case
        -- train0 selected for possible evolution
        RUNNING =0 &
        -- train0 actually allowed to move
        P0 < 6 &
        T0[P0+1] !=  T1[P1] &
        ...-- next place not occupied by other trains
        T0[P0+1] !=  T7[P7] & --
        -- critical section constraints satisfied
        RA + A0[P0+1] <= LA &
        RB + B0[P0+1] <= LB &  --
        RA + A0[P0+1] >= 0 &   --
        RB + B0[P0+1] >= 0     --
          -- RA updated according to movement of train0
        : RA + A0[P0+1];

           -- train1 selected for possible evolution
        RUNNING =1 &
          ... -- train1 actually allowed to move
           -- RA updated according to movement of train1
        : RA + A1[P1+1];
        ...
        -- no train can move
           (deadlock or all trains arrived)
        TRUE
          -- RA remains the same
        : RA;
      esac;
```

The description of the properties to be verified is expressed within the **CTLSPEC**/ **LTLSPEC** constructs of a NuSMV module. The property that all trains eventually complete their mission is encoded in the following way:

```
CTLSPEC -- all trains eventually complete their mission
   AF ((P0=6) & (P1=6) & (P2=6) & (P3=6) &
       (P4=6) & (P5=6) & (P6=6) & (P7=6))

LTLSPEC -- all trains eventually complete their mission
   F ((P0=6) & (P1=6) & (P2=6) & (P3=6) &
      (P4=6) & (P5=6) & (P6=6) & (P7=6))
```

The NuSMV version of the above CTL formula makes use of the same **AF** operator already seen in the previous Section. The only difference with respect to the UMC version is that now the state predicate to be verified is directly expressed in terms of values on internal variables of the model. The LTL version of the formula contains only the **F** operator applied to the same state predicate, because LTL formulas by definition must be satisfied by all the execution paths of the system (and cannot therefore contain further existential or universal quantifiers over the branches outgoing from the states). In this simple case it is quite immediate to understand that the two CTL and LTL formulas describe the same behavioural property.

Unfortunately, unless we introduce appropriate fairness constraints the above formulas would result to be *false*. Indeed, since the `next(P0)` function is total, a possible, infinite system evolution is the one in which only `train0` is selected for possible movement, i.e., during this evolution path the variable `RUNNING` is always equal to 0. In order to discard these uninteresting paths, and to make insignificant the dummy transitions corresponding to trains that are not moving, we must introduce a set of `FAIRNESS` constraints of the form:

```
FAIRNESS  RUNNING = 0;
    . . .
```

```
FAIRNESS  RUNNING = 7;
```

In this way, NuSMV limits its evaluations to the fair paths of the system evolutions, i.e. those infinite paths for which the fairness constraints are true for an infinite number of times. With the above constraints, an infinite path in which only `train0` is selected is discarded, because it violates the fairness rules `RUNNING=1, ..., RUNNING=7`.

If a logical formula is found to be false, NuSMV automatically returns a path as counterexample of the formula, and it is possible to check in detail the internal state of the variables for the states in the path. This approach works well for counterexamples of LTL formulas, which are just linear paths, but it does not work very well for counterexamples of CTL formulas which in general might have the form of a sub-graph of the system evolution graph. The task of understanding why a given counterexample path does not satisfy the expected property is left completely to user, i.e. no help is provided from the tool in understanding precisely why the evaluation failed. This does not constitute a problem in most cases, like our case, where the formula is rather simple and intuitive.

In our case, NuSMV found the formula to be true in about 413 seconds in the case of the CTL formula, and in about 166 seconds in the case of the LTL formula. However if the `RUNNNING` variable is declared as an *Input Variable* (**IVAR**) instead that as a *State Variable* (**VAR**), the execution times immediately decrease to 140 and 153 seconds respectively, and the CTL version not only recovers the original penality w.r.t. the LTL case, but even overtakes it.

Up to version 2.4 of NuSMV, a specific `process` construct was allowed to specify asynchronous systems. From version 2.5, this operator has been deprecated and it might be no longer supported in future versions of the tool. We have experimented also a specification of the model using the deprecated `process` construct. This alternative version is very similar the the current one, and essentially encloses the progression statements of the trains inside specific `process` modules. The evaluation time of this alternative version decreases to about 91 seconds in the CTL case and to about 88 seconds in the LTL case. This discrepancy in execution times is probably a sign of our relative inexperience in correctly using the tool and suggests that a deeper knowledge of the verification environment is needed for an actual mastering of the framework.

5 The Promela/SPIN Model

SPIN[5] [39] is an advanced and very efficient tool specifically targeted for the verification of multi-threaded software. The tool was developed at Bell Labs in the Unix group of the Computing Sciences Research Center, starting in 1980. In April 2002 the tool was awarded the ACM System Software Award. The language supported for the system specification is called Promela (PROcess MEta LAnguage). Promela is a non-deterministic language, loosely based on Dijkstra's guarded command language notation, and borrowing the notation for I/O operations from Hoare's CSP language. Once a model is formalised in Promela, a corresponding analyser is generated as a source C program (`pan.c`). The compilation and execution of the analyser performs all the needed on-the-fly state generations and verification steps. The properties to be verified can be expressed in LTL, and a violation of a property can be explained by observing the generated counterexample trail path.

In our case, a Promela model consists of (a) state variable declarations, (b) property specifications, and (c) system initialisation/execution code.

The state variables declarations (a) in our case consist in the definition of T_i, A_i, B_i vectors, plus the numeric variables P_i, RA, RB, LA, LB, as shown below.

```
//  mission data for train0
byte  T0[7];
 . . .
//  mission data for train7
byte  T7[7];
//  progress data for train0,...train7
byte  P0,...,P7;
// occupancy of region A
byte  RA;
// occupancy of region B
byte  RB;
// limit of region A
byte  LA;
// limit if region B
byte  LB;
// increments/decrements of train 0 for Region A
short A0[7];
 . . .
// increments/decrements of train 7 for Region A
short A7[7];
// increments/decrements of train 0 for Region B
short B0[7];
 . . .
// increments/decrements of train 7 for Region B
short B7[7];
```

The property (b) we are interested in is the classical property that all trains eventually complete their missions:

```
ltl p1 { <> ((P0==6) && (P1==6) && (P2==6) &&
            (P3==6) && (P4==6) && (P5==6) &&
            (P6==6) && (P7==6)) }
```

The above LTL formula is equivalent to the one already seen in the NuSMV example. The only difference is in the syntax of the *eventually* operator which is in this case encoded as **<>** instead of **F**.

The system initialisation/execution code (c) consists of: 1) the setting of the initial value for the state variables; 2) the possible activation of concurrent, communicating, asynchronous subprocesses (sharing the same

[5] `http://spinroot.com`

global memory); 3) the main execution of a sequence of statements. In Promela, sequences of statements, when included inside an `atomic {...}` construct, are executed as part of a single system (or process) transition.

The setting of the initial value for the state variables (1) has to assign a single numeric value to each vector component, as shown below:

```
init {
  atomic { //  initializations of state variable
   // T0:[1,9,10,13,15,20,23]
   T0[0]=1;  T0[1]=9;  T0[2]=10; T0[3]=13;
   T0[4]=15; T0[5]=20; T0[6]=23;
     . . .
   // T7:[26,22,17,18,12,27,8]
   T7[0]=26; T7[1]=22; T7[2]=17; T7[3]=18;
   T7[4]=12; T7[5]=27; T7[6]=8;
   A0[3]= 1; A0[5]= -1;      // A0:[0,0,0,1,0,-1,0]
     . . .
   A7[1]=1;  A7[4]=-1;       // A7:[0,1,0,0,-1,0,0]
   B0[3]=1;  B0[5]=-1;       // B0:[0,0,0,1,0,-1,0]
     . . .
   B7[0]=1;  B7[4]=-1;       // B7:[1,0,0,0,-1,0,0]
   RA=1;     RB=1;     LA=7;     LB=7;
  }   . . .// end of initializations of state variables
   . . . // activation of subprocesses
   . . . // main sequence of statements
}
```

In our case, we can avoid the definition and activation of subprocesses (2) – i.e. not modelling each train as a subprocess. Indeed, the non-determinism of the system can be modelled, as already done in the UMC and SMV case, by the non-determinism of the main process evolutions.

The main sequence of statements (3), in our case, is a loop of atomic guarded transitions, in which each transition models the progresses of a train.

```
init {
    . . .     //  initializations of state variables
   do       // main sequence of statements
    :: atomic {  // guarded progress of train0
     (// train0 has not yet completed its mission
      P0 < 6 &&
      T0[P0+1] != T1[P1] &&
      . . . // next place not occupied by other trains
      T0[P0+1] != T7[P7] &&
      // critical sections constraints satisfied
      RA+A0[P0+1] <= LA &&
      RB+B0[P0+1] <= LB
     ) ->
      // update the status of critical section A
      RA = RA + A0[P0+1];
      // update the status of critical section B
      RB = RB + B0[P0+1];
      // update the progress of train0
      P0++;
     };
     . . .
     :: atomic { // guarded progress of train1
         . . .
      };
     . . .
     // successful exit when all missions are completed
     :: (P0==6) && (P1==6) && (P2==6) && (P3==6) &&
        (P4==6) && (P5==6) && (P6==6) && (P7==6)
       -> break;
   od;
};
```

The evaluation of the formula is carried over by the process analyser (`pan.c`) in about 25 seconds, which decrease to 10 seconds when the process analyser is compiled with all `gcc` optimisations turned on (-O3 flag). We have also experimented the version of this specification in which each train was represented by an explicit process, whose activity consists in just executing the loop of its own atomic progress transition. This architecture, indeed, is the one which more precisely reflects our logical system design. In this case, the evaluation time raises to about 126 seconds (which decrease to about 47 seconds with `gcc` optimisations turned on).

Like in the case of NuSMV, when a formula does not hold it is possible to obtain a counter-example path to be analysed. Several features are explicitly provided for this purpose but we have experienced major difficulties in their use in terms of usability from the point of view of a non-experienced user.

6 The mCRL2 model

mCRL2[6][31] is a formal specification language with an associated toolset. The toolset can be used for modelling, validation and verification of concurrent systems and protocols. The mCRL2 toolset is developed at the department of Mathematics and Computer Science of the Technische Universiteit Eindhoven, in collaboration with LaQuSo, CWI and the University of Twente. The mCRL2 language is based on the Algebra of Communicating Processes (ACP) which is extended to include data and time. Processes can perform actions and can be composed to form new processes using algebraic operators. A system usually consists of several processes, or components, running in parallel. A process can carry data as its parameters. The state of a process is a specific combination of parameter values. In our case, we need to model the existence of a global status shared among the various trains, and this can be represented in mCRL2 by a single, recursive, non-deterministic process, whose parameters precisely model the global system state. Also in this case, the non-determinism of the system evolutions is modelled through the non-determinism of the main process behaviour.

In our case the mCRL2 specification includes (a) a data types specification; (b) actions specifications; (c) process definitions; (d) main process specification.

The data types specifications (a) in our case can be used to define the global constant data of our model. For example, we can model the vector of a train mission T_i as a `map`, i.e., a function from natural numbers (`Nat`) to natural numbers. The values returned by the function are expressed by means of the `eqn` construct.

[6] urlhttp://www.mcrl2.org/

```
map T0: Nat -> Nat;
   %% T0 [ 1, 9,10,13,15,20,23]
  eqn T0(0)=1; T0(1)= 9; T0(2)=10;
         ... ; T0(5)=20; T0(6)=23;
    . . .
map T7: Nat -> Nat;
   %% T7[26,22,17,18,12,27, 8]
  eqn T7(0)=26; T7(1)=22; T7(2)=17;
         ... ; T7(5)=27; T7(6)= 8;
```

Similarly, we can use the `map` construct for the critical sections limits (LA, LB), and for the vectors of increments A_i, B_i that trains should apply, with respect to critical sections, during their progress in the mission:

```
map  LA: Nat;          %%  limit for region A
 eqn  LA = 7;

map A0: Nat -> Int;
   %% A0 [0, 0, 0, 1, 0,-1, 0]
 eqn A0(0)=0; A0(1)= 0; A0(2)=0;
        ... ; A0(5)=-1; A0(6)=0;
    . . .
map B0: Nat -> Int;
   %% B0 [ 0, 0, 0, 1, 0,-1, 0]
 eqn B0(0)=0; B0(1)= 0; B0(2)=0;
        ... ; B0(5)=-1; B0(6)=0;
```

The actions specification (b) should define the structure of the possible actions (`act`) appearing inside processes. In our case, we define an action `move`, to represent the movement of the train at each progress step, and a final `arrived` action, which is performed when all trains have completed their missions:

```
act arrived; move: Nat;
```

The set of process definitions (c) consists in one unique recursive process, which we name `AllTrains`, whose parameters represent: (1) the progress indexes P_i of all the train missions, and (2) the occupancy counters of the two critical sections RA and RB.

```
proc AllTrains(P0:Nat, P1:Nat, P2:Nat, P3:Nat,
               P4:Nat, P5:Nat, P6:Nat, P7:Nat,
               RA:Int, RB:Int) =
   (P0 < 6               &&   % progress of train0
    T0(P0+1) != T1(P1)   &&
    . . .
    T0(P0+1) != T7(P7)   &&
    RA + A0(P0+1) < LA   &&
    RB + B0(P0+1) < LB
   )  ->  move(0).
    AllTrains(P0+1,P1,P2,P3,P4,P5,P6,P7,
              RA+A0(P0+1),RB+B0(P0+1))
  +
    . . .
  +
   (P7 < 6               &&   % progress of train7
    T7(P7+1) != T0(P0)   &&
    . . .
    T7(P7+1) != T6(P6)   &&
    RA + A7(P7+1) < LA   &&
    RB + B7(P7+1) < LB
   )  -> move(7).
    AllTrains(P0,P1,P2,P3,P4,P5,P6,P7+1,
              RA+A7(P7+1),RB+B7(P7+1))
  +   % all trains have completed their missions
  ((P0 ==6) && (P1 ==6) && (P2 ==6) &&
   (P3 ==6) && (P4 ==6) && (P5 ==6) &&
   (P6 ==6) && (P7 ==6)
   ) ->
    arrived . AllTrains(P0,P1,P2,P3,P4,P5,P6,P7,
                        RA,RB);
```

Finally, the main process specification (d) consists in the call of our `AllTrains` process with the appropriate initial data:

```
init   AllTrains(0,0,0,0,0,0,0,0, 1,1);
```

The mCRL2 toolset allows first to linearise the mCRL2 specification, and then to convert it into a linear process. Given a linear process and a formula that expresses some desired behaviour of the process, a PBES (Parametrised Boolean Equation System) can be generated. The tool `pbes2bool` executes the PBES and returns the evaluation status of the formula. The formulas supported by the mCRL2 toolset are based on full μ-calculus with parametric fix points.

The property that the system will eventually always reach a state in which all trains have completed their mission can be expressed as:

```
mu X.(([arrived] true) &&
          ([!arrived]X) && (<true> true))
```

The above formula is just a translation in μ-calculus of action-based CTL-like formula **AF** `{arrived} true` used with UMC. We refer to [19] and [31] for detailed description of the semantics of these two logics. The evaluation of this formula takes from 1 to about 19 minutes before returning the *true* value, depending on the options selected during the various evaluation steps. The greatest impact, which reduces the evaluation time from 19 minutes to about 1 minute and 40 seconds, is obtained with the selection of the `jittyc` data rewriting mode.

The logic supported by mCRL2 permits in many cases to replace the explicit use of fixpoints with the use of regular expressions inside box (`[...]`) and diamond (`<...>`) operators. Unfortunately that pattern is not applicable for the verification that the `arrived` event is always eventually reached. The simpler absence of deadlock can instead by checked with the formula `[true*]<true>true`.

When an unexpected *false* value is returned by the evaluation, the user can request the generation of a counterexample. This counterexample, however, is based on the structure of the evaluation process, and shows the occurred nested evaluations of the fixpoint formulas, without any link to the actual structure of the model or the details of its possible evolutions. The tool `lpsxsim` allows to explore the possible evolutions of the model under analysis. However, it does not seems that this exploration can be directly connected to a counterexample generated by a previous unsuccessful evaluation.

7 The FDR4 Model

FDR4[7] [28] is a refinement checker that allows to verify properties of programs written in CSP_M, a language that combines the operators of Hoare's CSP with a functional programming language. Originally developed by Formal Systems (Europe) Ltd in 2001, since 2008 is supported by by the Computer Sceince Department of University of Oxford. Being the specification approach based on a process algebra, the overall structure of the system is very similar to the one of MCRL2, i.e. we will have a single, recursive, non-deterministic, process definition whose parameters precisely model the global system state. The global data types and constants of our model are defined in a functional style. While sequences (encoded as `<value,...,value>`) are among the predefined data types, indexing inside them must be explicitly defined introducing a selector operator:

```
el(y,x)=
  if x==0 then head(y) else el(tail(y),x-1)
```

The global constants defining the train missions and region constraints can then be easily introduced as:

```
 ---- train missions -----
T0 = < 1, 9,10,13,15,20,23>
T1 = < 3, 9,10,13,15,20,24>
  . . .
T7 = <26,22,17,18,12,27, 8>

----- region A: train constraints ------
A0 = <0, 0, 0, 1, 0,-1, 0>
A1 = <0, 0, 0, 1, 0,-1, 0>
  . . .
A7 = <0, 1, 0, 0,-1, 0, 0>
LA = 7

----- region B: train constraints ------
B0 = <0, 0, 0, 1, 0,-1, 0>
B1 = <0, 0, 0, 1, 0,-1, 0>
 . . .
B7 = <1, 0, 0, 0,-1, 0, 0>
LB = 7
```

Also in this case we must declare the possible channel names appearing inside processes:

```
channel arrived, move
```

The recursive process definition, which we still name `AllTrains`, has as parameters the progress indexes P_i of all the train missions, and the occupancy counters of the two critical sections RA and RB.

```
AllTrains (P0,P1,P2,P3,P4,P5,P6,P7,RA,RB) =
     (P0 < 6 and          -- progress of train0
      el(T0,P0+1) != el(T1,P1)  and
        . . .
      el(T0,P0+1) != el(T7,P7)  and
      RA + el(A0,P0+1) <= LA and
      RB + el(B0,P0+1) <= LB
     )  & move ->
          AllTrains(P0+1,P1,P2,P3,P4,P5,P6,P7,
             RA+el(A0,P0+1),RB+el(B0,P0+1))
[]
       . . .
[]
  (P7 < 6 and           -- progress of train7
   el(T7,P7+1) != el(T0,P0)  and
     . . .
   el(T7,P7+1) != el(T6,P6)  and
   RA + el(A7,P7+1) <= LA and
   RB + el(B7,P7+1) <= LB
  )  & move ->
        AllTrains(P0,P1,P2,P3,P4,P5,P6,P7+1,
          RA+el(A7,P7+1),RB+el(B7,P7+1)
[]
 -- all trains have completed their missions
  (P0==6 and P1==6 and P2==6 and P3==6 and
   P4==6 and P5==6 and P6==6 and P7==6
  ) & arrived -> STOP
```

Finally, the main process specification consists in the call of our `AllTrains` process with the appropriate initial data, and with the hiding of the internal train moves:

```
SYS = AllTrains(0,0,0,0,0,0,0,0, 1,1)\{move}
```

The main difference of FDR4 with respect to all the previous approaches is that the system properties to be checked are specified not by means of temporal logics formulas, but by assertions stating adherence to a given abstract specification. In our case, for example, if we want to verify that the system always executes the `arrived` event, we must define an abstract specification like: `SPEC = arrived -> STOP` and then state that the system contains all the behaviours described by the specification:

```
assert SYS [FD= SPEC
```

and it does not introduce any further behaviour not described by the specification:

```
assert SPEC [FD= SYS
```

The concept of a system that *contains all the behaviours described by the specification* (and vice-versa) is not a trivial one, and can be adjusted according to several refinement notions, expressed by the [**T=** (Trace), [**F=** (Failure) and [**FD=** (Failure Divergences) refinement constructs. The most useful of these refinement notions is the [FD= refinement, which is the one used in the example. We refer to Hoare [38] and De Nicola *et al.* [14] for a deeper analysis of their relations and semantics.

While a case study with 6 trains instead of the usual 8 can be verified by FDR4 in about 15 seconds, the verification of the complete case with 8 trains took about one hour and 20 minutes. Probably a deeper knowledge of the framework and a more expert adjustment of its settings should allow to observe better performances.

[7] https://www.cs.ox.ac.uk/projects/fdr

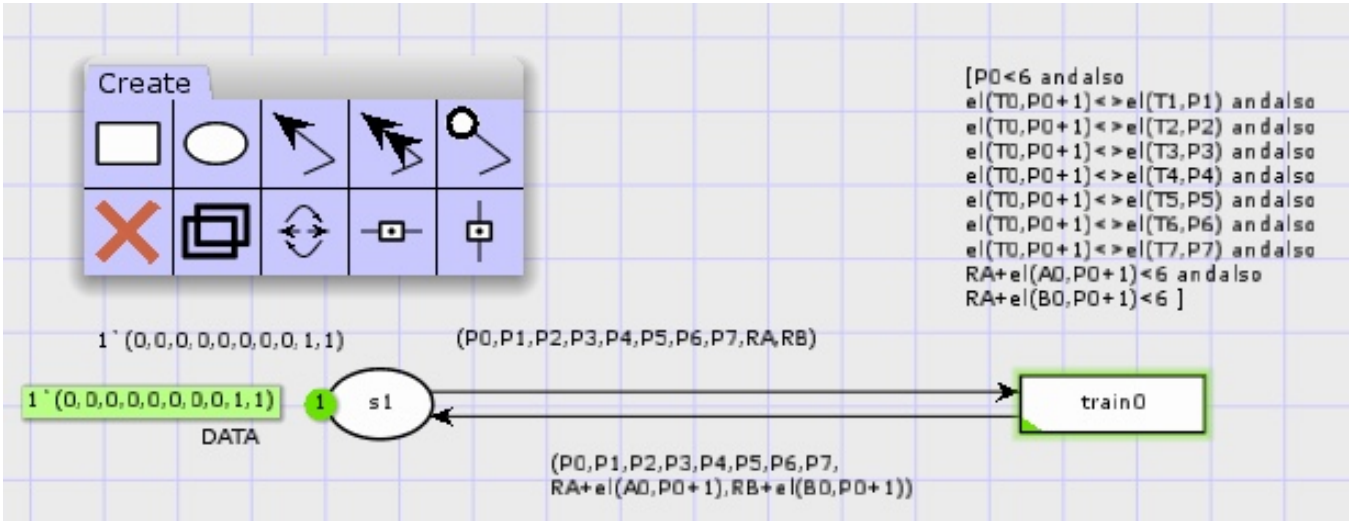

Fig. 3: A CPN transition modelling the activity of `train 0`

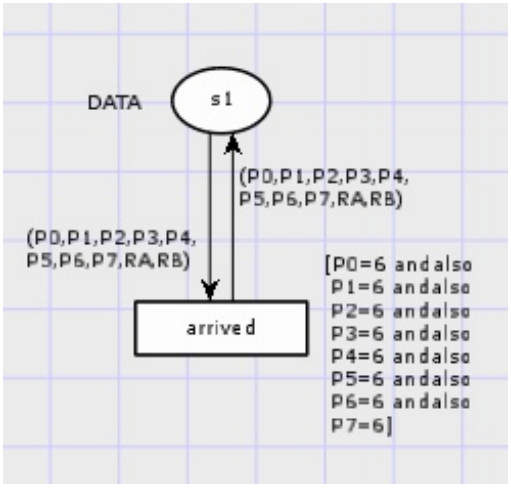

Fig. 4: Transition modelling the arrival of all trains

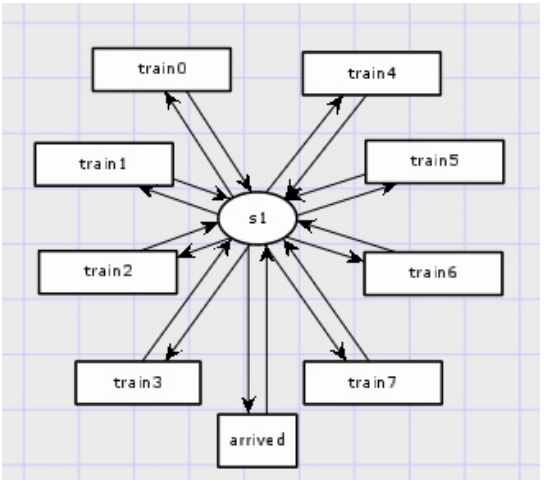

Fig. 5: The overall structure of the complete CPN

8 The CPN Tools Model

CPN Tools[8] is an environment for editing, simulating, and analysing Colored Petri Nets (CPN) [44]. It is originally developed by the CPN Group at Aarhus University from 2000 to 2010. The main architects behind the tool are Kurt Jensen, Søren Christensen, Lars M. Kristensen, and Michael Westergaard. From the autumn of 2010, CPN Tools is transferred to the AIS group, Eindhoven University of Technology, The Netherlands. The main difference between Coloured Petri Nets and ordinary Petri Nets is that the tokens that move across the network are allowed to contain some data (which *colour* them). Places of the network are typed with respect to the colour of the token they can contain. Transitions can be guarded with expressions that constrain that token allowed to pass, and may transform the data inside the token while moving from the source to the target place.

A direct mapping of our reactive model into a CPN can be achieved by modelling the system as a CPN with a single place `s1`, initially containing a single coloured token that represents the value of the initial system state. Outgoing transition from this place model the possible evolutions of the system: they (conditionally) accept the token from the source place, transform it according to the transition activity and return the modified token to its original place.

CNP Tools is a graphical tool, i.e., the CPN structure must be graphically drawn using *ad hoc* graphic tools. CPN places are represented by ovals, and CPN transitions elements by rectangles. The language used to describe the datatypes, the functions, and expressions is Standard ML, a powerful functional language which is also at the base of the FDR4 tool.

[8] `http://cpntools.org`

Figure 3 shows the CPN transition modelling the activity of `train0`. The label of the edge that exits from place `s1` is labelled with an expression that describes the structure of the system state: as for all the previous cases, the data consists of the sequence containing the various train progress indexes P_i and current occupancy counters for the two critical regions `RA` end `RB`. The inscription associated to the guard of transition `train0` describes the conditions under which the transition is allowed to fire, and these are precisely the same conditions already seen in all the previous cases. The label of the edge returning to the source place `s1` describes the transformation performed by the transition on the current system state, and corresponds precisely to the usual transformation performed by the activity of `train0`.

Figure 4 shows the CPN transition modelling the reaching of the final status of the system, when all the trains have completed their missions. Apart from its graphical notation, the information is also in this case the same as in all the previous cases.

The overall structure of the resulting CPN – omitting all inscriptions and labels – is shown in Figure 5.

The first step of the verification of a CPN network consists in completely generating its state-space. Once that is done it is possible to observe a generation report, which, among other information about pre-defined properties, states how many deadlock states have been generated. According to the documentation it should also be possible to evaluate CTL formulas, expressed as functions written in the ML language, on the overall system behaviour. Unfortunately, we have not been able to actually activate this CPN Tools functionality, due to the difficulty in understanding and putting into practice the related documentation, which appeared scattered in different documents, and, to the best of our understanding, incomplete.

The main problem found with this tool is its performance during the state-space generation. While the state-space of a system with 4 trains can be generated in about 14 seconds, the state-space of a system with 6 trains requires 9 minutes. We have not been able to generate the state-space for the complete case with 8 trains.

9 Tool Comparison

The pattern of having a set of global data that is concurrently and atomically updated by a set of concurrent guarded agents [15] is a formalisation pattern often encountered also in the railway field. In our case, we met this pattern during the verification of the deadlock avoidance kernel inside the ground scheduling system that controls the movements of driverless trains inside a given yard. This pattern can be rather easily formalised and verified using different languages and frameworks. We have experimented with six possible alternatives, i.e., UMC, NuSMV, Promela/SPIN, mCRL2, CPN and FDR4, which differ greatly in maturity, support alternative verification logics, and provide different degrees of friendliness and flexibility in the user support during the formalisation and verification steps.

The activity is still in progress, since, on the one hand, we plan to extend our experiments to several other well known toolsets, and, from the other hand, there are still many aspects of the currently explored frameworks that need a deeper understanding and evaluation. Notwithstanding the preliminary nature of our experiments, it is useful to report a comparison of the different tools, in which we discuss the features offered by the environments, based on six broad parameters that had an impact on our experience, namely (1) specification formalism, (2) property definition language, (3) documentation, (4) platforms compatibility, (5) user friendliness, and (6) performance. The parameters have been evaluated by the authors in the context of the current experience, and, although the evaluation is biased by our background and by the specific context of this work, we believe that it can offer a useful perspective on the applicability of the tools to the railway context. More specifically, the evaluation is given by authors who have experience with the UMC tool, state-machines based formalisms, and CTL logic, while they are novice users for the other tools and languages. In the following paragraphs, we describe the parameters, and, based on them, we compare the different tools, while in Table 1 we summarise our evaluation. Part of the parameters, e.g., specification language and property definition language, are rather objective. Instead, part of them, e.g., documentation and user friendliness, are subjective, and are arguably evaluated by the authors according to a 3-point Likert scale, with three qualitative grades, namely (L)ow, (M)edium and (H)igh. The justification for the different grades are given in the paragraphs associated to each parameter.

Specification Language The reference family of the language supported by a tool to specify the model is a parameter that a designer should carefully consider when choosing a formal environment. Indeed, based on (a) the confidence that the designer has with a certain formalism, and (b) the type of problem at hand, the modelling activity can be extremely fluid, or particularly cumbersome. The deadlock avoidance algorithm could be easily represented with the different languages, but it is useful to report the general differences among the tools considered. Three families of specification languages can be observed, namely state-machine oriented representations, process algebras, and Petri Nets. Among the three families, the state-machine oriented representation, which allows an explicit shared data structure, is the most intuitively suitable for the formalisation of our problem, in which agents atomically read and update a common data blackboard. Indeed, with process algebras, a sin-

Table 1: Comparison between the different environments in the context of the presented experiments.

	Specification Language	**Property Definition Language**	**Doc.**	**Platform Compatibility**	**User Friendliness**	**Performance**
UMC	State Machines (Structured Data)	μ-Calculus CTL/ACTL	L	Online, Unix, Windows, macOS	H	H
NuSMV	State Machines (Flat Data)	CTL/LTL/PSL	H	Unix, Windows, macOS	H	H
Promela/SPIN	State Machines (Flat Data)	LTL	M	Unix, Windows, macOS	M	H
mCRL2	Process Algebra (Structured Data)	parametric μ-Calculus	H	Unix, Windows, macOS	M	M
CPN Tools	Petri Nets (Structured Data)	CTL	M	Windows	L	L
FDR4	Process Algebra (Structured Data)	Refinement Checking	H	Unix, Windows, macOS	H	L

gle recursive process is used for our purposes, and the system state is represented as a process parameter (see Sect. 6 and 7). Process algebras are more oriented to model designs with communication agents that do not share a global status. Instead, with Petri Nets, the system state was concealed in the colour of a token (Sect. 8). Also in this case, our model with a single place is not the intuitive way of modelling with Petri Nets, which are a more natural choice when one wants to model the flow of a set of activities. It shall be noticed that, in our context, we were interested in *replicating* the same design solution, initially specified in UMC, with the different tools. Different results might be obtained if one starts from, e.g., a Petri Nets-oriented specification in which the railway layout is explicitly represented.

Another observation related to the specification languages concerns the data structures made available by the different environments. NuSMV and Promela/SPIN admits only integer and vector types of fixed size[9], and do not allow for complex or more dynamic data structures. Instead, UMC admits also dynamically sized vectors, with nested vector data structures. The remaining tools have the full power of functional languages, allowing for complex data types, including high-order types (e.g., functions as data).

Property Definition Language The language in which a property can be expressed affects the type of properties that can be verified on a certain design. In our case, the deadlock property is rather simple, and can be easily expressed with the different languages supported by the tools. However, it is useful to briefly summarise the languages supported, since, in some cases, not all properties can be verified by all the tools, and this may impact on the choice of the formal environment to adopt.

The most powerful environment in terms of property definition language is mCRL2, which supports a parametric version of μ-Calculus. Instead, UMC supports plain μ-Calculus, which still subsumes both CTL and LTL. However, most of the properties that one encounters in practice are more intuitively expressed directly in CTL and LTL. Therefore, although from theoretical viewpoints those tools that support solely CTL or LTL are less powerful, they may be easier to use in practice. FDR4 is not based on temporal logic, but uses a refinement checking approach, in which the property to be verified is represented with the same specification language of the model. From the pointof view of the user, this allows to avoid learning temporal logics. On the other hand, building a correct specification of the intiuitive property that one wishes to verify might not be straightforward.

Documentation The quality of the documentation, in terms of amount, readability, and degree of update, are crucial aspects for the usability of a formal tool, especially in a context in which the user does not have a tool expert "on call", who can support him/her during the usage. With some exceptions, the quality of the documentation is generally acceptable for all the tools considered, except for UMC, which does not have an up-to-date documentation for the current version[10]. The quality of the documentation for Promela/SPIN and CPN Tools has been indicated as medium (M), due to the complexity of the documentation content and structure, which is, however, available and up-to-date. In addition, for CPN Tools, we were not able to find the necessary instructions to verify CTL properties.

Platform Compatibility Although not having a direct impact on the usability of a tool, its compatibility with multiple platforms gives an indication of the potential audience of a formal environment. Indeed, while operating system (OS) emulators exist that can support soft-

[9] NuSMV allows solely for constant vectors.

[10] The update of the documentation is in progress

ware developed for different OSs, a user might not even start using a tool simply because it is not supported by his/her preferred OS, or the OS used by the company. With the exception of CPN Tools, all the considered environments are available on all the platforms. All our experiments were performed on macOS, and, in case of CPN tools, a Windows emulator was used. Therefore, we do not know whether all the tools offer the same features in the different platforms. Strictly connected with platform compatibility is the complexity of the installation procedure. Indeed, given the large availability of diverse formal tools, if the installation procedure is not perceived as quick, and easy enough by the user, s/he might decide that the software is not usable even before starting to use it. In our case, we did not encounter difficulties in installing the different tools. Overall, we can say that the preliminary, and potential, obstacles in starting to use the tools, are successfully addressed by all the environments.

User Friendliness After installing and having learned the basics of a tool by reading its documentation, an aspect to consider is how easy it is, for a novice, to learn to use the platform in a proficient way for the problem at hand. Almost all platforms seem to be mainly tailored to the (successful) validation of a (correct) system design. When it comes to providing the user with an easy-to-understand description of *why* a given system design does not behave as expected, some of the tools show losses in terms of usability. While with UMC, NuSMV and FDR4, we did not encounter problems in the usage, and in identifying design errors, issues emerged with the other platforms. Promela/SPIN is intrinsically complex, and given our background, mostly based on CTL, switching to a tool based on the LTL paradigm requires more effort. Instead, with mCRL2 we had difficulties in interpreting the counterexamples, since it appeared particularly difficult to make sense of the link between the property evaluation results and the operational system behaviour. Finally, several usability problems were encountered with the GUI of CPN Tools. In particular, the GUI does not have an intuitive behaviour, e.g., the process of selecting/deselecting widgets works with a toggle-button paradigm, which complicates a fluid interaction. In addition, the incompleteness of the documentation did not help in taking out the best from the tool. We also admit that we are not experts with CPN, and this might have biased our evaluation.

Performance Fig. 6 summarises the execution time for each model configuration adopted in our experiments[11]. We do not show the performance of CPN Tools, since we were not able to complete the verification process for the 8-trains model. Already from the data that we have collected we can observe that apparently small choices in the construction of the models, or in the selection of the best options for the evaluations, can greatly affect the performance of the verification. Almost all the tools show extremely great differences in terms of execution times depending on the choices done by the user. In our case, we obtained the best performances from NuSMV by declaring the RUNNING variable as **IVAR**. For SPIN, the usage of the -O3 flag (i.e., all `gcc` optimisations turned on) was the factor determining the major decrease in terms of verification time. For mCRL2, the selection of the `jittyc` data rewriting mode was crucial in increasing the performance. Finally, with UMC, the lower verification time was obtained with a parallel version of the tool, and selecting the non-interactive evaluation mode. It is worth noticing the tools with a more powerful specification and property definition languages also suffer from weaker performance with respect to simpler tools. More specifically, mCRL2 appear to have lower performance w.r.t. Promela/SPIN, NuSMV and UMC, but, still, our design is verified in a reasonable amount of time (2s). Instead, FDR4 and CPN Tools appear not to be able to handle intrinsically complex problems of large size, as the one considered in our study, in an efficient way.

The differences in terms of verification times obtained, and the different solutions adopted for each tool to minimise the verification time, indicate that a deep mastering of the tools is required to exploit at their best the capabilities of the various frameworks.

10 Towards Formal Methods Diversity

An interesting consideration that has been stimulated by our experimentation is that modelling and verifying a system using different approaches can really give a plus in the reliability of the verification results. We have actually experienced that the effort of modelling and checking a system design in several variants is essential for identifying the errors introduced in the construction of the formal specification or in the verification process. The possibility to model and verify a certain design with completely different verification frameworks can be an interesting solution also from the point of view of the *validation* of critical systems. Indeed, while none of the verification tools considered is designed and validated at the greatest safety integrity levels by itself, the existence of different, non validated, tools producing the same result might increase the overall confidence on the verification results. This observation poses the basis for a novel concept for railways, which is *formal methods diversity*. The idea is to apply the concept of diversity, quite common in safety-critical systems engineering [45, 53], in the application of formal methods. More specifically, we suggest to use different non-certified formal environments for the modelling of verification of a certain

[11] Each configuration corresponding to a point in Fig. 6, with associated verification time, is available at `http://fmt.isti.cnr.it/WEBPAPER/STTT2017data.zip`.

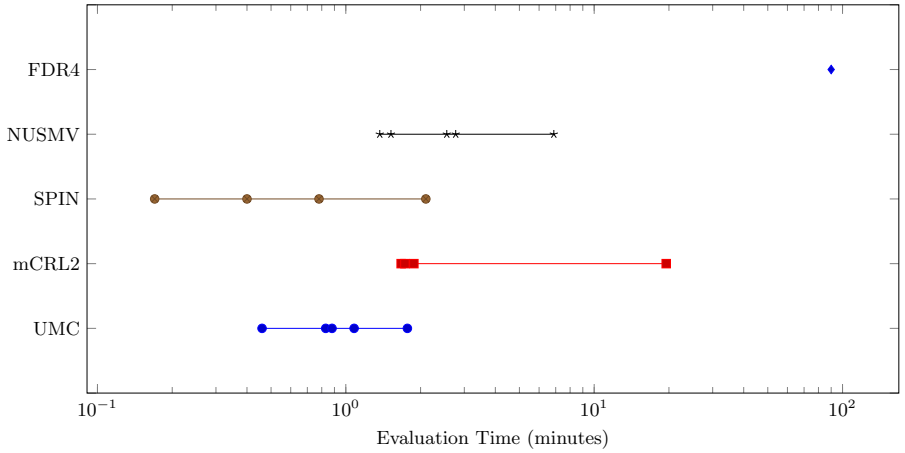

Fig. 6: Summary of evaluation time ranges for the frameworks (logarithmic scale)

railway problem or design, and compare the results. Of course, this simple concept has possible hurdles in terms of applicability. Below, we reflect on the potentials and challenges that the idea opens, based on our experience and knowledge of the railway industry.

Specification Validation In the experience described in this paper, we validate the *specification*[12] of an algorithm, by ensuring that different formal environments produce the same verification results. The same idea can be applied whenever one has developed a specification for a certain system, and wishes to translate it into different frameworks, to increase the reliability of the verification results. The translation can be performed manually, as in our case, or automatically, as performed by Rockwell Collins in the avionic domain [51]. Regardless of the means used for translation, the errors that might raise in this context are: (a) errors in the specification, which may be introduced in the design phase by the system designer; (b) errors in the translation of the specification, introduced by the automatic or human translators; (c) errors concealed in the environments used for formal modelling and verification, since, as observed, being the environments themselves not certified, some of them might include errors that can be revealed only when the results of the verification differ from those of other environments.

Diversity in Properties One of the potentials offered by the usage of different environments is associated to the diversity of logics that the environments support for the definition of properties to be verified. In our context, we used properties that can be equivalently specified with CTL and LTL logics, but, as well know, the two logics are not comparable [12], and different requirements might have forms that can be specified only with one logic. Therefore, the availability of diverse environments gives also the possibility of verifying properties that have, e.g., an inherent CTL nature, with CTL-oriented environments, and properties that have an inherent LTL nature with LTL-oriented ones. In this sense, formal methods diversity also enlarges the scope of properties that can be verified for the same specification. A wider analysis on properties that are typical of the railway domain and that can be verified with the different tools, as performed, e.g., by Frappier *et al.*[26] in the context of information systems verification, will clarify to which extent formal methods diversity can facilitate the verification of railway systems.

Requirements Validation Formal methods diversity can be applied also if one wishes to pursue *requirements* validation [9], e.g., to check completeness and consistency, instead of specification validation as in our experience. In this case, one should use different formal environments to provide alternative specifications for the same requirements. In a requirements validation context, we argue that employing the same formal methods expert for the modelling tasks is not recommended, since s/he might be biased towards a certain architecture, and might replicate the same, potentially erroneous, design decisions in the different specifications. In addition, different formal environments might give different modelling capabilities, and one might not use them at their best if s/he is biased towards the replication of the same specification. This opens to the possibility of diversifying formal methods experts, as it happens when different developers are employed to implement software variants [3, 45]. This choice of having different models designed by different experts has to be handled with care, since it may trigger complications in further development stages. Indeed, if only one specification is chosen for a single implementation, one might partially loose the benefits of modelling diversity.

[12] The concept of specification is intended here in Jackson's terms [33], i.e., the model that, given certain environmental assumptions, shall satisfy the requirements.

On the other hand, if also code diversity is employed [3], with each implementation being derived from different specifications, modelling diversity can be exploited at its full benefits. This observation suggests that, when formal methods diversity is adopted, also the overall railway development process shall be adapted. This is an issue that we have previously encountered in railways when passing from a code-centered development paradigm to a model-centered development one, in which code generation was used [21]. Formalising a railway process, adherent to the CENELEC EN 50128 norm [8], and based on formal methods diversity is beyond the scope of this paper. However, this formalisation becomes mandatory when one wishes to apply the approach in the railway industry.

Knowledge and Experience with Formal Environments As already emphasised, one of the major hurdles in applying formal methods diversity is the experience required to proficiently handle different formal environments. In our case, we showed that if a designer is proficient with a formal tool, in our case UMC, s/he can use this experience to successfully translate a model into other formal tools with limited effort. On the other hand, design decisions affect the *performance* of the tools, as we have shown, e.g., for NuSMV (Sect. 4). Therefore, if one is oriented to exploit the capabilities of different tools at their best, higher proficiency is required with different tools. This goal is practically hampered by the lack of documentation of part of the tools considered, an issue that deeply affects the possibility of effectively learning different tools with acceptable effort. This aspect can be mitigated by employing multiple experts of different environments, but we know that, from an industrial perspective, this requires a dedicated, or outsourced, formal methods group, and, more in general, a major uptake of formal methods by railway practitioners [25].

Appropriateness of a Formal Tool for a Design We have seen that our algorithm design can be represented with six different tools, but this might not be true for all the railway-specific problems. Hence, particular care and guidance is required in the choice of the formal framework to adopt in order to model and verify the specification [61]. For example, in the literature we see that state-based graphical specifications are used to model the control logic of ATC/ATP systems [22,40,54,10,20], while interlocking systems are often modelled with textual specification, and verified by means of model checking [59,58,37,23,7,42]. A clear definition of guidelines for the choice of the *appropriate* formal method, or set of formal environments, to be used for a specific railway problem is therefore required to make formal methods diversity applicable. Further practical and comparative research, as the one performed, e.g., by Zave [61] in the context of network protocols, shall be performed in the railway domain to achieve this goal.

Evolution and Acceptance of Formal Tools The tools that we used in our experiments are freely available, and mostly maintained by universities or public organisations. Even within the time span in which this paper was written, evolution in terms of versions of the tools was observed (e.g., FDR 4.2.0 was released on December, 2016). Keeping the pace of the evolution of a single tool is complex, and it requires to rely on a robust framework of release control, which ensures backward compatibility of the platforms, and forward compatibility of the artifacts created with the platforms. The problem becomes even more complicated if one company has to follow the evolution of multiple environments at the same time, as required if formal methods diversity is applied. The development of a railway system can take several years, continuous updates might be required by the customer, and one has to rely on stable tools versions. In addition, in our experience [22], railway companies are keen to prefer commercial tools, rather than open source ones, also for the availability of assistance. We are aware that, in general, the open source world is evolving towards a business model in which the revenues come through the assistance services offered for free, or commercial versions, of the tools. Hence, we foresee that, if this business model gets a foot hold for formal environments, also the mindset of railway companies might be more open to these tools, and formal methods diversity has some additional chance to become an established practice in the railway industry.

11 Conclusion

The world of formal methods offers several options in terms of automated environments [26,17], which can and have been used to verify the design of railway systems [18, 29]. In this paper, we show the application of six different formal tools, namely UMC, Promela/SPIN, NuSMV, mCRL2, CPN Tools and FDR4, in the modelling and verification of a deadlock avoidance algorithm for train scheduling [49]. The algorithm takes care of avoiding situations in which a train cannot move because its route is blocked by another train. This is a typical problem, which can be modelled according to a blackboard architectural pattern [15], in which concurrent guarded agents atomically update a global data blackboard. Our experience shows that limited effort is required to adapt the same design to different formal environments. This observation opens up new possibilities for the establishment of the concept of *formal methods diversity* in railways. The idea is that the application of diverse, non-certified formal tools on the same design allows to increase the confidence on the correctness of the verification results. The paper compares the characteristics of the different tools, in light of our modelling and verification experience, and discusses the industrial potential

and challenges associated to the application of formal methods diversity in the railway context.

Our personal interest is now to further experiment with additional free and open source tools, such as LTSMin [6], LTSA [46], DiVinE [4], JavaPathFinder [36], Alloy [41], and commercial tools, such as CADP [27], SCADE [16], and Stateflow with Simulink Design Verifier [35]. Our idea is to model our prototypical railway problem with these different tools, to have a more complete in-field understanding of the practical hurdles that formal methods practitioners may face when dealing with diverse formal methods.

References

1. Accellera, property specification language - reference manual - version 1.01, 2003.
2. Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.
3. Algirdas Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.
4. Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0–an explicit-state model checker for multithreaded C & C++ programs. In *International Conference on Computer Aided Verification*, pages 863–868. Springer, 2013.
5. Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor: A successful application of b in a large project. In *International Symposium on Formal Methods*, pages 369–387. Springer, 1999.
6. Stefan Blom, Jaco van de Pol, and Michael Weber. Ltsmin: Distributed and symbolic reachability. In *International Conference on Computer Aided Verification*, pages 354–359. Springer, 2010.
7. Andrea Bonacchi, Alessandro Fantechi, Stefano Bacherini, Matteo Tempestini, and Leonardo Cipriani. Validation of railway interlocking systems by formal verification, a case study. In *International Conference on Software Engineering and Formal Methods*, pages 237–252. Springer, 2013.
8. CENELEC. EN 50128:2011: Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. Technical report, 2011.
9. Angelo Chiappini, Alessandro Cimatti, Luca Macchi, Oscar Rebollo, Marco Roveri, Angelo Susi, Stefano Tonetta, and Berardino Vittorini. Formalization and validation of a subset of the european train control system. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 109–118. IEEE, 2010.
10. Chan-Ho Cho, Dong-Hyuk Choi, Zhong-Hua Quan, Sun-Ah Choi, Gie-Soo Park, and Myung-Seon Ryou. Modeling of cbtc carborne ato functions using scade. In *Control, Automation and Systems (ICCAS), 2011 11th International Conference on*, pages 1089–1093. IEEE, 2011.
11. Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.
12. Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.
13. Clara DaSilva, Babak Dehbonei, and Fernando Mejia. Formal specification in the development of industrial applications: Subway speed control system. In *Proceedings of the IFIP TC6/WG6. 1 Fifth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols: Formal Description Techniques, V*, pages 199–213. North-Holland Publishing Co., 1992.
14. Rocco De Nicola and Matthew CB Hennessy. Testing equivalences for processes. *Theoretical computer science*, 34(1-2):83–133, 1984.
15. Jing Dong, Shanguo Chen, and J-J Jeng. Event-based blackboard architecture for multi-agent systems. In *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, volume 2, pages 379–384. IEEE, 2005.
16. Francois-Xavier Dormoy. Scade 6: a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS08)*, pages 1–9, 2008.
17. Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
18. Alessandro Fantechi. Twenty-five years of formal methods and railways: what next? In *International Conference on Software Engineering and Formal Methods*, pages 167–183. Springer, 2013.
19. Alessandro Fantechi, Stefania Gnesi, Alessandro Lapadula, Franco Mazzanti, Rosario Pugliese, and Francesco Tiezzi. A logical verification methodology for service-oriented computing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(3):16, 2012.
20. Alessio Ferrari, Alessandro Fantechi, Stefania Gnesi, and Gianluca Magnani. Model-based development and formal methods in the railway industry. *IEEE software*, 30(3):28–34, 2013.
21. Alessio Ferrari, Alessandro Fantechi, Stefania Gnesi, and Gianluca Magnani. Model-based development and formal methods in the railway industry. *IEEE Software*, 30(3):28–34, 2013.
22. Alessio Ferrari, Alessandro Fantechi, Gianluca Magnani, Daniele Grasso, and Matteo Tempestini. The metrô rio case study. *Science of Computer Programming*, 78(7):828–842, 2013.
23. Alessio Ferrari, Gianluca Magnani, Daniele Grasso, and Alessandro Fantechi. Model checking interlocking control tables. In *FORMS/FORMAT 2010*, pages 107–115. Springer, 2011.
24. Alessio Ferrari, Giorgio O Spagnolo, Giacomo Martelli, and Simone Menabeni. From commercial documents to system requirements: an approach for the engineering of novel cbtc solutions. *International Journal on Software Tools for Technology Transfer*, 16(6):647–667, 2014.

25. John Fitzgerald and Peter Gorm Larsen. Balancing insight and effort: The industrial uptake of formal methods. In *Formal methods and hybrid real-time systems*, pages 237–254. Springer, 2007.
26. Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of model checking tools for information systems. In *International Conference on Formal Engineering Methods*, pages 581–596. Springer, 2010.
27. Hubert Garavel, Radu Mateescu, Frédéric Lang, and Wendelin Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In *International Conference on Computer Aided Verification*, pages 158–163. Springer, 2007.
28. Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew W Roscoe. Fdr3a modern refinement checker for csp. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201. Springer, 2014.
29. Stefania Gnesi and Tiziana Margaria. *Formal methods for industrial critical systems: A survey of applications*. John Wiley & Sons, 2012.
30. Stefania Gnesi and Franco Mazzanti. An abstract, on the fly framework for the verification of service-oriented systems. In *Rigorous software engineering for service-oriented systems*, volume 6582 of *LNCS*, pages 390–407. Springer, 2011.
31. Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and analysis of communicating systems*. 2014.
32. Stefan Gruner, Apurva Kumar, and Tom Maibaum. Towards a body of knowledge in formal methods for the railway domain: Identification of settled knowledge. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 87–102. Springer, 2015.
33. Carl A Gunter, Elsa L Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.
34. Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
35. Grégoire Hamon, Bruno Dutertre, Levent Erkok, John Matthews, Daniel Sheridan, David Cok, John Rushby, Peter Bokor, Sandeep Shukla, Andras Pataricza, et al. Simulink design verifier-applying automated formal methods to simulink and stateflow. In *AFM08: Third Workshop on Automated Formal Methods 14 July 2008 Princeton, New Jersey*, 2008.
36. Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.
37. Anne E Haxthausen. Automated generation of formal safety conditions from railway interlocking tables. *International journal on software tools for technology transfer*, 16(6):713–726, 2014.
38. Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.
39. Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.
40. Simon Hordvik, Kristoffer Øseth, Jan Olaf Blech, and Peter Herrmann. A methodology for model-based development and safety analysis of transport systems. In *11th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2016.
41. Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.
42. Phillip James, Andy Lawrence, Faron Moller, Markus Roggenbach, Monika Seisenberger, Anton Setzer, Karim Kanso, and Simon Chadwick. Verification of solid state interlocking programs. In *International Conference on Software Engineering and Formal Methods*, pages 253–268. Springer, 2013.
43. L. Jansen, M. Meyer Zu Horste, and E. Schnieder. Technical issues in modelling the European Train Control System (ETCS) using Coloured Petri Nets and the Design/CPN tools, 1998.
44. Kurt Jensen and Lars M Kristensen. *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media, 2009.
45. G Latif-Shabgahi, Julian M Bass, and Stuart Bennett. A taxonomy for software voting algorithms used in safety-critical systems. *IEEE Transactions on Reliability*, 53(3):319–328, 2004.
46. J Magree. Behavioral analysis of software architectures using ltsa. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 634–637. IEEE, 1999.
47. Franco Mazzanti. An experience in Ada multicore programming: parallelisation of a model checking engine. In *Ada-Europe International Conference on Reliable Software Technologies*, volume 9695 of *LNCS*, pages 94–109. Springer, 2016.
48. Franco Mazzanti, Alessio Ferrari, and Giorgio O Spagnolo. Experiments in formal modelling of a deadlock avoidance algorithm for a cbtc system. In *International Symposium on Leveraging Applications of Formal Methods*, pages 297–314. Springer, 2016.
49. Franco Mazzanti, Giorgio Oronzo Spagnolo, Simone Della Longa, and Alessio Ferrari. Deadlock avoidance in train scheduling: a model checking approach. In *International Workshop on Formal Methods for Industrial Critical Systems*, volume 8718 of *LNCS*, pages 109–123. Springer, 2014.
50. Franco Mazzanti, Giorgio Oronzo Spagnolo, and Alessio Ferrari. Designing a deadlock-free train scheduler: A model checking approach. In *NASA Formal Methods Symposium*, volume 8430 of *LNCS*, pages 264–269. Springer, 2014.
51. Steven P Miller, Michael W Whalen, and Darren D Cofer. Software model checking takes off. *Communications of the ACM*, 53(2):58–64, 2010.
52. Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.
53. David Powell, Jean Arlat, Ljerka Beus-Dukic, Andrea Bondavalli, Paolo Coppola, Alessandro Fantechi, Eric Jenn, Christophe Rabéjac, and Andrew Wellings. Guards: A generic upgradable architecture for real-time dependable systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):580–599, 1999.

54. Jie Qian, Jing Liu, Xiang Chen, and Junfeng Sun. Modeling and verification of zone controller: the scade experience in china's railway systems. In *Complex Faults and Failures in Large Software Systems (COUFLESS), 2015 IEEE/ACM 1st International Workshop on*, pages 48–54. IEEE, 2015.
55. Maurice H ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 76(2):119–135, 2011.
56. Maurice H ter Beek, Stefania Gnesi, and Franco Mazzanti. From EU projects to a family of model checkers. In *Software, Services, and Systems*, volume 8950 of *LNCS*, pages 312–328. Springer, 2015.
57. Somsak Vanit-Anunchai. Application of coloured petri nets in modelling and simulating a railway signalling system. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 214–230. Springer, 2016.
58. Linh Hong Vu, Anne E Haxthausen, and Jan Peleska. Formal modelling and verification of interlocking systems featuring sequential release. *Science of Computer Programming*, 133:91–115, 2017.
59. K Winter, W Johnston, P Robinson, P Strooper, and L Van Den Berg. Tool support for checking railway interlocking designs. In *Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55*, pages 101–107. Australian Computer Society, Inc., 2006.
60. Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):19, 2009.
61. Pamela Zave. A practical comparison of alloy and spin. *Formal Aspects of Computing*, 27(2):239, 2015.

View publication stats