

Towards Formal Methods Diversity in Railways: an Experience Report with Seven Frameworks

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. In the ever expanding universe of formal methods, several 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 report in formal modelling and verification using seven different formal environments, namely UMC, Promela/SPIN, NuSMV, mCRL2, CPN Tools, FDR4 and CADP. 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 set of global data is atomically updated by a set of concurrent guarded agents. Our experience, limited to the specific problem, shows that the design of the algorithm can be translated into the different formalisms with acceptable 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. Furthermore, by checking that the number of states generated during the verification process is the same for each framework, the designer can have an initial indication of the equivalence of the diverse models. 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 language features and performance, and dis-

cusses 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 [13], 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 [24, 39, 71]. The usage of the B method [2] for the development of the SACEM system – a control platform for a line of Paris RER [18] –, and the iterative formal verification of the Paris automatic metro line 14, also based on the B method [8], 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 [17], experiences on the application of these approaches were performed in railways, especially for what concerns the validation of interlocking systems [70, 68, 44, 29, 11, 49, 4]. More recently, formal model-based approaches [69, 26], 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 [28, 47, 63, 15, 27]. Some experiences were also performed on the usage of Coloured Petri Nets (CPN) for modelling and simulation of railway signalling platforms [67, 50]. Recently, experiences have been published in which model checking and

induction-proof techniques are used in combination for the verification of several railway systems [9].

When using any support tool (e.g., compilers, testing environments, formal verification frameworks) along the development of a railway product, the CENELEC EN 50128 norm asks the tool to be qualified, or certified, for its usage in the process [13]. This requirement is common to other standards, as, e.g., the DO 178C for the software of avionic systems [64]. Although formal tools exist that are certified according to the EN 50128 norm, as, e.g., SCADE [22] 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 formal 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 on the verification outcomes. This principle was previously applied in the avionic domain by Rockwell Collins [60], which, in collaboration with other partners, developed translators from semi-formal models expressed in Simulink/Stateflow towards the Lustre formal language [41], and then towards formal environments, such as PVS [61] and NuSMV [16], 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 seven different tools, namely: UMC [65], Promela/SPIN [46], NuSMV [16], mCRL2 [38], FDR4 [35], Coloured Petri Nets (CPN) Tools [51] and CADP [33]. We have selected model checking tools, given the increasing interest in this technology shown by the railway sector in the last years [24]. 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 [57, 58] of a Communications-based Train Control System (CBTC) [30]. 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 [21] – which can be modelled as a set of global data that is 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 to as the *blackboard* architectural pattern [21]. In this paper, we show the design of the algorithm, the different models produced with the seven formal tools, and the results of the verification activities, observing differences and hurdles in the usage of the seven environments. All the models produced within this experience, and referred in this paper, are available in our public repository [56].

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 shows that, given a simple blackboard system design, while limited effort and adjustments were required to translate the design into different formalisms, a much greater effort was needed to fully exploit the various verification framework capabilities. Small choices in the specification of the models, or in the verification options, resulted in a great impact on the performance of the tools. Our goal is to ensure that, given a certain specification, different non-certified formal tools provide the same verification results. In this way, although the tools are not certified, we can increase the confidence on the correctness of the specification. From his point of view our main focus is on the validation of the specification rather than on the validation of the requirements – see Sect. 12 for more details. We also suggest a lightweight method to provide an initial indication on the equivalence of the specifications designed with the different tools, which is based on observing the number of states produced by the formal tools. If the number of states is the same, and all the specifications satisfy the properties, this increases the confidence on the equivalence of the specifications. To fully ensure specification equivalence, model transformation and verification of the translation step [3] should be performed.

Our proposal is focused on the railway domain, given the interest of the domain in formal methods [24], and the certification constraints [13]. Nevertheless, the presented principles, which take inspiration from code/design diversity [12, 52, 62], and early studies on diversity of formal approaches [5], can in principle be applied also to other domains.

The paper extends a previous contribution to the ISoLA 2016 conference [56]. With respect to this previous work, the current one describes the experience with three additional environments, namely CPN, FDR4 and CADP (Sect. 7, Sect. 8 and Sect. 9), provides a more in-depth discussion on the lessons learned while using the seven tools (Sect. 11), and discusses the potential of formal methods diversity in the railway domain (Sect. 12).

The rest of the paper is structured as follows. In Sect. 2 we describe the deadlock avoidance algorithm that we modelled. In Sect. 3–9, we show our models and the verification results for UMC, NuSMV, Promela/SPIN,

mCRL2, FDR4, CPN Tools and CADP, respectively¹, and, within the descriptions of the models, we highlight the peculiarities of the different languages and environments. In Sect. 10 we present a more complex case, based on an extension of the original design, in which trains perform round-trip missions. All the models referred in this paper can be retrieved from the data repository [59]. In Sect. 11, we provide a discussion on the experience, and in Sect. 12 we discuss the potentials and the challenges associated to the concept of formal methods diversity. Finally, Sect. 13 concludes the paper and discusses our future work.

2 The Deadlock Avoidance Algorithm

This section describes basic elements of the modelled algorithm, which was defined in our previous works [57, 58]. 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 the identification of the endpoint in which the train is at a certain moment. We will have 8 variables P_0, \dots, P_7 , one for each train, which store the current index for the train. For example, at the beginning, we have $P_0 = 0, \dots, P_7 = 0$, 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 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 [57, 58]. 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 system state logically corresponds to a set of train progresses and each train movement logically corresponds to an atomic system evolution step, leads to a state-space of 1,636,535 configurations. This data is important because it will allow the user to cross-check the correctness of the encoding of this logical design in the various frameworks.

3 The UMC Model

UMC [65] is a model checker that belongs to the KandISTI² [66] 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 interface³.

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 [37]. The graph is formalised as an abstract doubly labelled transition system (L2TS) [20]. Labels on the states represent the observable properties of the system states, and labels on the edges represent the observable properties of the atomic system transitions. The logic supported by the KandISTI framework uses the evolution graph as semantic model and allows the user to specify abstract properties in a way that is rather independent from the internal implementation details of the system [25]. From this point of view the state

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)

² <http://fmt.isti.cnr.it/kandisti>

³ <http://fmt.isti.cnr.it/umc>

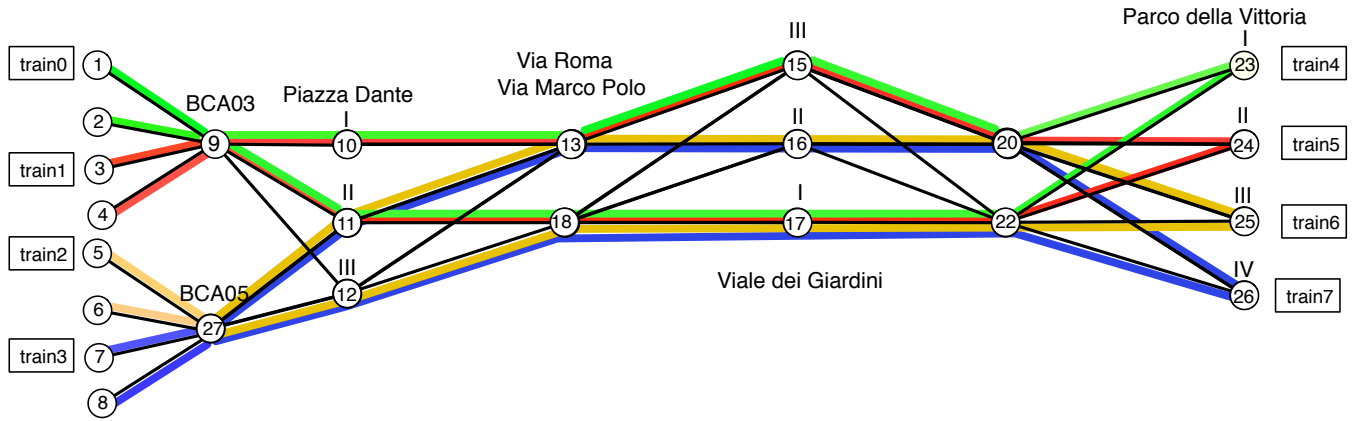


Fig. 1: A fragment of the yard layout and the 8 missions of the trains

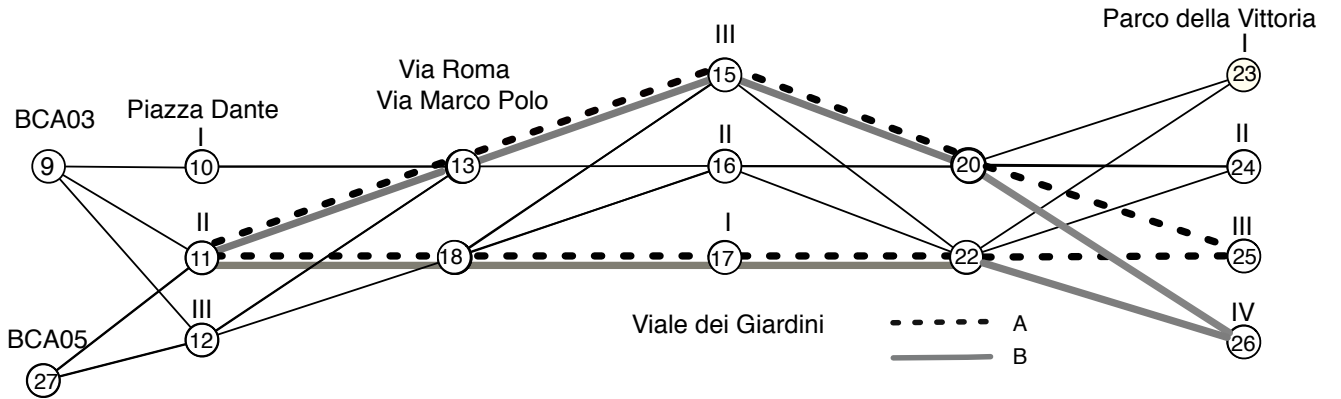


Fig. 2: The critical section A and B which must not be saturated by 8 trains

labels become the the state predicates of the logic, and action labels become the basic actions of the logic.

The different flavours of the various tools of KandISTI family are related to the supported specifications languages, that range from process algebras to sets of UML-like statecharts. In our case, we will use the UMC tool (version 4.6), because it allows the user to model in a direct way nondeterministic system evolutions (triggered by global conditions) that read and update global data. Moreover UMC is the only tool of the KandISTI family that supports composite data structures.

It is not the main concern of paper is give a detailed presentation of UMC, for which we refer to the specific documents available online⁴. Here we focus instead on those aspects used by our models. 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 behaviour.

The Vars Part 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 pre-

vious vectors –, the occupancy counters RA and RB of the two critical 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];
. . .
-- RA updates steps for train0
A0: int[] := [0, 0, 0, 1, 0, -1, 0];
. . .
-- RA updates steps for train7
A7: int[] := [0, 0, 0, -1, 0, 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[] := [0, 0, 0, -1, 0, 0, 0];
-- occupancy of region RB
RB: int :=1;
-- train progresses
P0, P1, P2, . . . , P7: int :=0;

```

⁴ <http://fmt.isti.cnr.it/umc/DOCS>

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 ⁵ highly dynamic. This dynamism might lead to potentially infinite state systems.

The Behavior Part 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:

```
Behavior:
<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 sequentially 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 train `T0`, 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;
}
```

Verification As a last step we have to define what we want to see on the abstract L2TS associated to the system evolutions. Indeed, we recall that the overall behaviour of a system is formalised as an abstract L2TS,

⁵ i.e. the sum of the sizes of the current values held by all variables.

and abstraction rules allow us 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 in which 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
}
```

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 provides an explanation of why the evaluation of the formula failed, allowing the user to interactively explore the set of system evolution steps that led to failure of the relevant subformulas and view all the internal details of the traversed states.

UMC completes the evaluation of the formula returning *true* in a time that ranges from 38 seconds to 86 seconds depending on how the tool is used. The fastest results of 38 seconds is obtained by exploiting a multi-core approach during statespace generation[55], and by adopting a depth-first exploration strategy.

Cyclic Generalisation The case study illustrated above is a particularly simple model, in which a set of trains perform a limited one way mission across a yard. In general the situation can be more complex, e.g. with trains that repeatedly perform one mission after another, continuously cycling across the yard. In this case the evolution graph would contain: (a) *fair* cycles in which all the trains always eventually move even if they might never pass again from a state in which all the trains are in their final destination at the same time (b) *bad* cycles in which a few trains actually block each other while the system as a whole would continue to evolve with the

non deadlocked trains (i.e. a case of partial deadlock); (c) *not relevant* cycles in which only a few trains evolve, but just because of allowed unfairness of the underlying dispatching policy.

Under these circumstances the above formula `AF ARRIVED` can no longer be used to evaluate the correctness of the model because it would signal as errors all the above three cases of cycles. There is another CTL property that allows us to distinguish a correct model from a wrong one, which is represented by the formula:

```
AG EF ARRIVED
```

The above formula states that from every reachable state of the system (AG) there is at least one path (EF) that leads to a state in which all trains have reached their final destination (ARRIVED). This formula is false only in presence of true partial deadlocks, in which some trains are no longer allowed to reach their destination, independently from the fairness of the dispatching policy. An additional benefit of the the above formula is that, if violated, the corresponding explanation provided by UMC would show a precise path towards the train movement that is the *real* cause of a *possibly future* partial deadlock. Let us consider, for example, the case of two trains trying to traverse the same linear sequence of itineraries in opposite directions. The real cause of the deadlock would be the entering of the second train inside that linear sequence, while the explicit partial deadlock would occur at a later time when the two trains would actually meet face to face. A full deadlock would occur when no more trains in the system were allowed to move.

4 The NuSMV Model

NuSMV⁶ [16] 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 the user 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 structures of basic data types. A state of the system is represented by a set of variables. Assignment rules in the language allow the user to specify *total* functions, which define all the possible values that a state variable can assume in the next state.

Constants and Variables 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:

⁶ <http://nusmv.fbk.eu/>

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,-1, 0, 0, 0];
LB := 7;
B0 := [0, 0, 0, 1, 0,-1, 0];
. . .
B7 := [0, 0, 0, -1, 0, 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` input 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.

IVAR

```
RUNNING: 0..7;
```

VAR

```
P0: 0..6;
. . .
P7: 0..6;
RA: 0..8;
RB: 0..8;
```

Behaviour The initial state of the system can be described within the `ASSIGN` construct making use of the `init` operator:

ASSIGN

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

The *total* transition relation that models all the possible system evolutions can be defined within the `TRANS` construct structured using a nested sequence of conditional expressions (`condition? thenpart: elsepart`) as shown by the following rule:

TRANS

```
-- progression rule for the evolving train T0
RUNNING =0 &
-- the train has not yet completed its mission
P0 < 6 &
--
-- the next place is not occupied by other trains
T0[P0+1] != T1[P1] &
T0[P0+1] != T2[P2] &
T0[P0+1] != T3[P3] &
T0[P0+1] != T4[P4] &
T0[P0+1] != T5[P5] &
T0[P0+1] != T6[P6] &
T0[P0+1] != T7[P7] &
--
-- the progression step satisfies all constraints
RA + A0[P0+1] <= LA &
RB + B0[P0+1] <= LB
```

?

```

-- T0 advances one step
next(P0) in (P0+1) &
next(P1) in P1 &      -- ot
-- the other trains do not move
next(P2) in P2 &
next(P3) in P3 &
next(P4) in P4 &
next(P5) in P5 &
next(P6) in P6 &
next(P7) in P7 &
-- critical sections occupancy is updates
next(RA) in (RA + A0[P0+1]) &
next(RB) in (RB + B0[P0+1])
:
-- progression rule for the evolving train T1
RUNNING =1 &
. . .
:
-- progression rule for the evolving train T7
RUNNING =7 &

```

Since the rule must be total, we must add a final `:elsepart` describing the system transition in the case the train selected by the `RUNNING` input value is not allowed to move. In these cases the system status should not change.

```

: -- no train can move
next(P0) in P0 &
. . .
next(P7) in P7 &
next(RA) in RA &
next(RB) in RB

```

Verification 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. However, unless we introduce appropriate fairness constraints the above formulas would appear to be *false*.

In fact, the final else clause of the transition relation, triggered when an input-selected train cannot move, introduces non progressing self loops in the system evolutions. In order to discard these uninteresting paths, and to make insignificant the dummy transitions corresponding to trains that are not allowed to move, 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`. With the introduced `FAIRNESS` constraints, we find the formula to be true in about 39 seconds in the case of the CTL formula, and in about 43 seconds in the case of the LTL formula. It is worth noticing that, in the UMC model presented in Sect. 3, fairness issues did not arise, because all paths are finite⁷.

A more efficient way of verifying with NuSMV the correctness of the system behaviour is to avoid the introduction of the `FAIRNESS` constraints and verify instead the NuSMV CTL equivalent of the already seen `AG EF ARRIVED` property (i.e. that from any state the system has the capability to reach the successful final state). In this case the property rule becomes:

```

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

```

Indeed in this case the formula is proved to be true in just 2.8 seconds.

When a logical formula is found to be false, NuSMV automatically returns a path as counterexample of the formula, in the shape of an evolution trace, and it is possible to check in detail the internal values of the variables along the states in the path. Since in general the counterexample for a branching time formula might have the shape of a tree, the returned path would necessarily describe just a fragment of the real counterexample.

Cyclic Generalisation We have already seen in the UMC case that the verification of the branching time formula `AG EF ARRIVED` allows us to verify the correctness of the cyclic model (i.e. find livelocks) also in presence on unfair scheduling paths. This alternative formula has also the advantage of identifying the *real* cause of partial deadlocks as soon as they are triggered and, in the NuSMV case, the effect of making unnecessary the addition of `FAIRNESS` assumptions, with great advantages in terms of performance.

5 The Promela/SPIN Model

SPIN⁸ [46] (Simple Promela Interpreter) is an advanced and very efficient tool specifically targeted for the veri-

⁷ The language used by UMC does not support explicit fairness constraints. Instead, fairness-related properties can be specified by means of the supported logics, e.g., μ -Calculus.

⁸ <http://spinroot.com>

fication 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 the Promela model consists in single main process which defines a set of global state variables, their initialisations, and a main execution body. A Promela model can also include a set of property specifications that will be verified by the generated process analyser.

State Variables 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 train T0 ... T7
byte T0[7], ... ,T7[7];
// progress data for train0,...train7
byte P0, ...,P7;
// constraints of train T0 ...T7 for Region A
short A0[7], ... , A7[7];
// constraints of train T0 ...T7 for Region B
short B0[7], ... , B7[7];
// occupancy of region A, B
byte RA, RB;
// limits of region A, B
byte LA, LB;
```

Initialisation The system initialisation appears within the `atomic {...}` construct inside the system `init {...}` section. 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 has to assign a single numeric value to each vector component, as shown below:

```
init {
  // initializations of state variables
  atomic {
    // 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]
    T7[0]=26; T7[1]=22; T7[2]=17; T7[3]=18;
    T7[4]=12; T7[5]=27; T7[6]=8;

    // A0:[0,0,0,1,0,-1,0]
    A0[3]= 1; A0[5]= -1;
```

```
. . .
// A7:[0,1,0,-1,0,0,0]
A7[1]=1; A7[3]=-1;

// B0:[0,0,0,1,0,-1,0]
B0[3]=1; B0[5]=-1;
. . .
// B7:[0,0,0,-1,0,0,0]
B7[3]=-1;

RA=1; RB=1; LA=7; LB=7;
}

// main sequence of statements
. . .
}
```

Behaviour In our case, 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, in our case, is a `do` loop containing a sequence of atomic guarded transitions, in which each transition models the progress rule for a train. The `do` loop still appears inside the `init` section, after the initialization code.

```
init {
  // initializations of state variables
  . . .
  // main sequence of statements
  do
    :: atomic {
      // progress rule for train0
      (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 sections
      RA = RA + A0[P0+1];
      RB = RB + B0[P0+1];
      // update the progress of train0
      P0++;
    };
    . . .
    :: atomic {
      // progress rule for train7
      . . .
    };
  // all missions are completed
  :: (P0==6) && (P1==6) && (P2==6) && (P3==6)
    && (P4==6) && (P5==6) && (P6==6) && (P7==6)
    -> skip;
  od;
};
```

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

```
ltl p1 {eventually ((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 `eventually` instead of `F`.

The evaluation of the formula is carried out by the process analyser (`pan.c`) in about 13 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. In this case, the evaluation time raises to about 47 seconds. The introduction of processes with the only purpose of replacing an internal sequential nondeterminism with an external interprocess nondeterministic scheduling does not seem to pay off from the point of view of the performance. Surely also in the case of SPIN a more detailed fine tuning of the many options provided by the tool would allow us to further increase its overall performance.

When a formula does not hold the tool saves a counterexample trail which can be further analysed. Since the logic supported by SPIN is a linear time logic, it is not possible to express and verify branching time properties like `AG EF ARRIVED`. This means that in the cyclic extension of the model we might have difficulties in proving, for example, the absence of livelocks.

6 The mCRL2 model

mCRL2⁹[38] 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.

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. Our mCRL2 specification includes a set of data types specification which describe the constants of our model, a set of actions specifications that qualify the possible kinds of system evolution steps, a single process definition and a main process specification.

Data Types Specifications The data types specifications in our case are used to define the vectors of the train missions, the vectors of the sections constraints, and the limits associated to each critical section. In particular we have modeled 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.

```
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 have used 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;
```

Actions Specification The actions specification should define the structure of the possible actions 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;
```

Process Definitions The set of process definitions consists in one unique recursive process, which we name `AllTrains`, whose parameters `P0, ..., P7` represent the progress indexes P_i of all the train missions, while the `RA, RB` parameters represent the occupancy counters of the two critical sections A and B . The body of this process definition specifies the nondeterministic rules that govern the train evolutions in the usual way.

```
proc AllTrains(P0:Nat, P1:Nat, P2:Nat, P3:Nat,
              P4:Nat, P5:Nat, P6:Nat, P7:Nat,
              RA:Int, RB:Int) =
    % progress of train0
    (P0 < 6          &&
     T0(P0+1) != T1(P1) &&
     . . .
```

⁹ <http://www.mcrl2.org/>

```

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))
+
. . .
+ % progress of train7
(P7 < 6          &&
 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);

```

Main Process Specification Finally, the main process specification 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);
```

Verification The mCRL2 toolset allows us 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] X) && (<true> true)
```

The evaluation of this formula takes from 2 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 2 minutes, is obtained with the selection of the `jittyc` rewriting option that compiles the rewriting engine to be used for the evaluation of the formula¹⁰. Further minor optimizations are surely possible but it is outside the purpose of the paper to analyse all of them.

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. For example the absence of deadlock can be checked with the formula `[true*]<true>true`.

The mCRL2 framework has no problems in verifying also the other branching time formula – equivalent to `AG EF ARRIVED` – which can be expressed as:

```
nu X.(![true*.arrived>true] && ([true]X))
```

or, using regular expressions:

```
[ true* ] < true* . arrived > 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 the user to explore the possible evolutions of the model under analysis. However, it does not seem that this exploration can be directly connected to a counterexample generated by a previous unsuccessful evaluation.

Model Variants We have also made several experiments in which the global status of the system was modeled by explicit processes instead that as data argument of the unique system process. In particular we have experimented the use of (i) one process for each itinerary endpoint, modelling its occupancy state, (ii) one process for each critical section, modelling its availability, and (iii) one process for each train, modelling its mission. The overall system is now resulting by the parallel composition of all these components that appropriately synchronize to model the desired system evolutions. All the systems described in this way (we have tried several versions of them) result much less performing than our initial nondeterministic sequential case (the execution times range from 78 minutes to a few hours) and are therefore not further discussed.

7 The FDR4 Model

FDR4¹¹ [35] is a refinement checker that allows the user to verify properties of programs written in `CSPM`, 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 the Computer Science 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.

¹⁰ http://mcl2.org/web/user_manual/tools.html

¹¹ <https://www.cs.ox.ac.uk/projects/fdr>

Data Types and Constants The global data types and constants of our model are defined in a functional style. While sequences (encoded as $\langle \text{value}, \dots, \text{value} \rangle$) 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 the region constraints can 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
```

Recursive Process Definition 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
```

Main Process Specification 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}
```

Verification 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 can define an abstract specification like: `SPEC = arrived -> STOP` and then state that the system is a valid refinement of the specification.

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

The concept of a system that *refines the behaviour described by the specification* 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 [45] and De Nicola *et al.* [19] 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.

When a refinement assertion returns a negative result, a system evolution trace is also presented as counterexample. This trace is just a sequence of communication actions, with no information about the structure of the CSP processes at the various steps, and in the case of nondeterministic models it might not be trivial to understand precisely which synchronizations actually occurred during the trace (even if we reveal the underlying actions hidden behind the top level tau actions).

Model Variants Also in this case we have experimented several different designs of the system that better exploit the compositional features of the framework. In particular, as already tried in the mCRL2 case, we have modeled the system as a parallel composition of processes, using one process for each itinerary endpoint, one process for each critical section and one process for each train.

The result has been quite amusing as the verification of the system with 8 trains has now been carried out in about in a few tens of seconds (from 28 to 41 seconds depending on the chosen design alternatives) versus the about 80 minutes of the sequential version. We omit here

the presentation of the various encoding of the parallel versions, which can however be found among the examples in the data repository of all the experiments [59].

Let us now take into consideration the other branching time property, i.e. that from any reachable state it should be possible to reach a target state in which all trains are at their target destination. It becomes very difficult in this case to find a specification – in terms of CSP processes – and refinement relation which allows us to check whether a system verifies this property.

For example, we might adopt as specification a process that behaves as shown in Fig. 3, but there is no refinement relation which can distinguish a correct implementation from the wrong implementation shown in Fig. 4. If we consider our more general example of continuously cycling trains (see Sect. 3), this means that we might have some difficulty in discovering partial deadlocks – i.e., cases in which the state `arrived` becomes nomore reachable even if some of the trains are still allowed to continuously move.

8 The CPN Tools Model

CPN Tools¹² is an environment for editing, simulating, and analysing Colored Petri Nets (CPN) [51]. 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. The outgoing transitions 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 transition 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.

Data and Behaviour Figure 5 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 6 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 7. A reader with experience in CPN may observe that the presented model is a counter intuitive formalization of the problem. However, we recall that we made an effort in faithfully translating our initial specification into the different languages, and, to replicate our UMC model, we had to chose this modelling style. Different styles may be chosen, if the interest is in requirements validation instead of specification validation, as in our case – see Sect. 12.

Verification 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 write and evaluate ML functions that perform some standard queries and new specific computations on the underlying system evolutions graph. For example the expression `NoOfNodes()` allows the user to see the number of states of the system, and `NoOfSecs()` shows the time taken to generate the state space. The nodes of the state space are consecutively numbered and the initial state has number 1. The internal details of the status of a system configuration (i.e. the marking of a node) can be seen by evaluating the expression `print(NodeDescriptor n)`, where `n` is the node number. The expression `ListDeadMarkings()` lists of all the nodes without successors, while the expression `ListHomeMarkings()` lists all the nodes that are reachable by all the nodes of the state space. If our model is correct we would have precisely one home marking constituted by the state in which all the trains are arrived at their destination. In case of problems the list of home markings would be *null*. To see why an error situation has occurred we should first find the internal number of the node corresponding to the expected final node (i.e. the node in which the

¹² <http://cpntools.org>



Fig. 3: Spec AG EF arrived

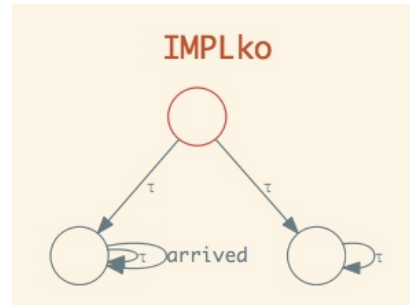


Fig. 4: Partially deadlocking implementation

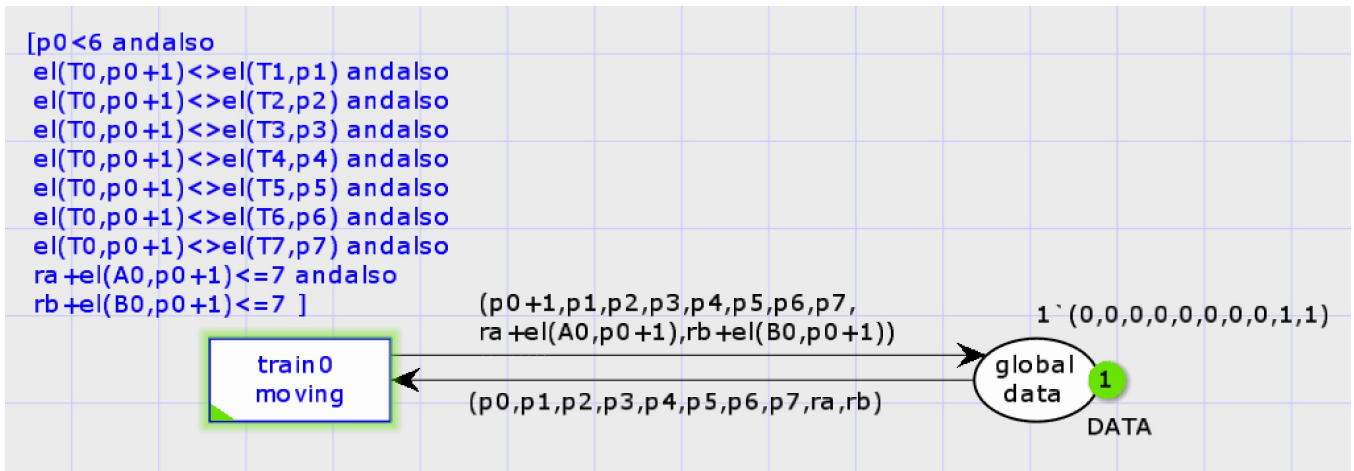


Fig. 5: A CPN transition modelling the activity of train 0

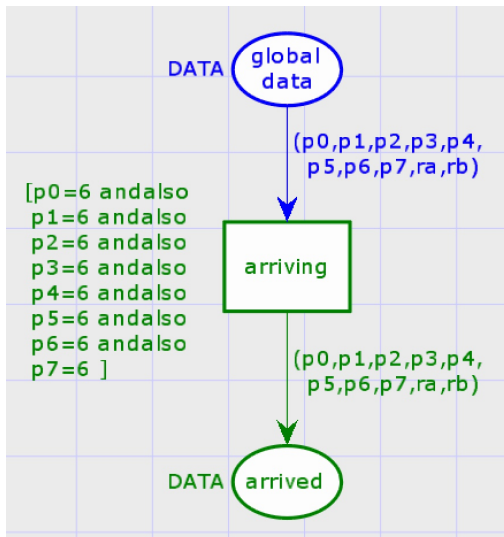


Fig. 6: Transition modelling the arrival of all trains

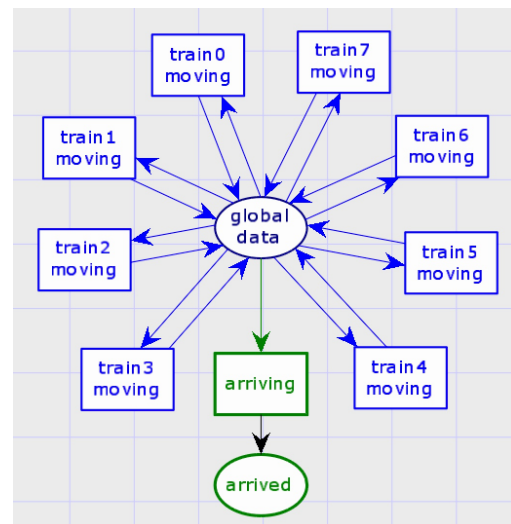


Fig. 7: The overall structure of the complete CPN

place `arrived` contains 1 token), and then find all the nodes of the statespace from which that final node is not reachable. The evaluation of the expression:

```
SearchNodes(EntireGraph,
  fn n => (length(Mark.SYS'arrived 1 n) =1),
  NoLimit , fn n => n, [], op::);
```

displays as result (in the 6 trains case):

```
val it = [60272] : Node list
```

The above result indicates that the list of nodes satisfying our requests contains precisely one node identified by number 60272.

We can now search the state space for any node from which our final 60272 node is not reachable, by evaluating the expression:

```
SearchNodes(EntireGraph,
  fn n => (not (Reachable (n, 60272))),
  NoLimit , fn n => n, [], op::);
```

The displayed result is:

```
val it = [] : Node list
```

and it indicates that the list of the nodes in which a partial deadlock occurs is empty.

If we evaluate the ML expression `Reachable' (1, xx)`; we would get a list of nodes (i.e. that path) that connect node 1 (the initial state) with node `xx` (that can be the final or a dealock state).

Notice that the property we are verifying is actually the one of the kind `AG EF ARRIVED`, which allows us to find the errors also in the general case of continuously cycling trains.

It is also possible, by loading an ad-hoc `ASK_CTL` package to write and evaluate ML expressions that correspond to CTL-like formulas. No proofs / counterexamples / explanations are however generated after the evaluation.

The main problem found with this tool is its performance during the state-space generation. While the state-space of a system with 5 trains (10410 states) requires 14 seconds, the statespace of a system with 6 trains (60272 states) requires about 9 minutes. We have not been able to generate the statespace for the complete case with 7 trains (323196 states) even after 12 hours of execution.

Model Variant In order to see if this performance problem was caused by our rather particular use of the Petri Net tool, which stressed the use of coloured tokens versus the usual place/ transitions constructs, we have also tried to redesign the system using a normal Petri Net without making use of coloured tokens. The adopted structure, here shown for modelling the activity of just one train, is the one illustrated in Figure 8.

In that structure, we have used one place for each itinerary endpoint, that initially contains a token only if the endpoint is not occupied by a train. We have used one place for each critical section, that initially contains as many tokens as the number of trains allowed to enter the critical section. Finally we have a set of places modelling the current progress of the train, that initially contains a token if that train is at that stage of the progress. The transitions of the system are constituted by the transitions of the trains that can move from one step to the next one only if the next endpoint is not occupied and possibly there are no problems in entering any critical section required by step. The effect of the transition is to make unavailable to other trains the next endpoint, to release to other trains the previously occupied endpoint, and possibly to remove or reintroduce a token in the places representing the critical sections. While the performance of this pure Petri Net system is slightly higher, (e.g. 6.5 minutes versus the original 9 minutes in the case of 6 trains) the tool is still not able to generate the full statespace for systems with an higher number of trains¹³.

9 The CADP Model

CADP¹⁴ (Construction and Analysis of Distributed Processes) is a verification framework for the design of asynchronous concurrent systems [33]. While its origins dates back to the mid 80s, since than it has been continuously improved and enriched, and is currently actively maintained by the CONVECS team at INRIA. It has been used in many industrial projects among many different application fields. Among the various languages supported for the specification/design or models, we have chosen the LNT (Lotos New Technology)[34] notation which, having an imperative style, is the one that better reflects our style of design. For the evaluation of the system properties we have selected the Evaluator4 model checking engine, that allows to verify formulas written in MCL (Model Checking Language)[54]. MCL is a powerful branching time temporal logics extends a regular alternation-free μ -Calculus with data-handling, regular formulas over transition sequences, and fairness operators. In our case a LNT model consists in module defining a set of types for the various data elements used in the model, plus a single sequential nondeterministic process that contains all the system data and evolves according the rules associated to the train movements.

¹³ CPN Tools requires a Windows system. We made our experiments both on a Windows Virtual Machine running under macOS with 64GB RAM, and on a dedicated Windows system with 64 GB RAM. In both cases the used memory remained far below the available memory provided by the System. The “CPN Tools State Space Manual” says that 200,000 nodes is the upper limit for the size of state spaces.

¹⁴ <http://cadp.inria.fr/>

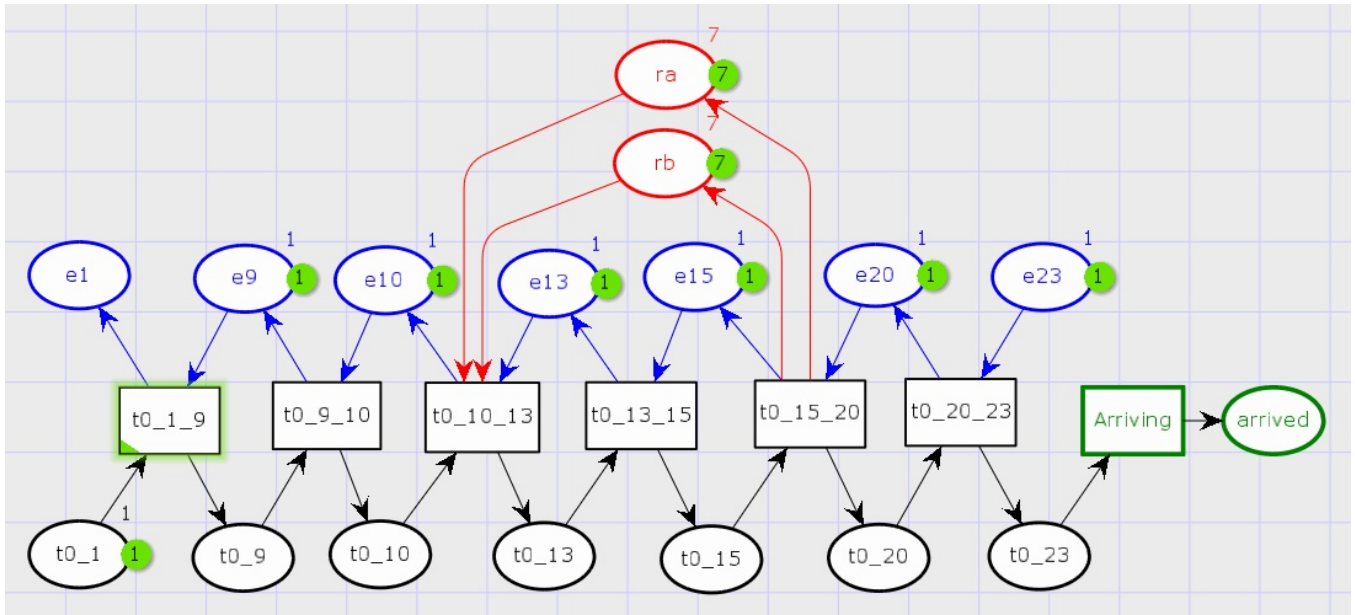


Fig. 8: The basic Petri Net modelling the activity of train 0

Data Types Specifications The data types specifications assign a name and a definition to each class of values used in the model. In particular, we have:

```

type Train_Number is
  range 0 .. 7 of nat
end type

type Train_Mission is
  array [0 .. 6] of nat
end type

type Train_Constraint is
  array [0 .. 6] of int
end type

channel Movement is
  (Train : Train_Number)
end channel

```

Process Definition The system behaviour can be described by a single nondeterministic process that executes a main loop which includes the nondeterministic choice among all the trains allowed to progress. A final clause of the choice is triggered when all the trains have completed their missions.

```

process MAIN [MOVE: Movement, ARRIVED: none] is
  var P0,P1,P2,P3,P4,P5,P6,P7 : nat,
      RA,RB : int,
      LA,LB : int,
      T0,T1,T2,T3,T4,T5,T6,T7 : Train_Mission,
      A0,A1,A2,A3,A4,A5,A6,A7 : Train_Constraint,
      B0,B1,B2,B3,B4,B5,B6,B7 : Train_Constraint
  in
    -- data initializations
    P0 := 0; . . . ; P7 := 0;
    RA := 1; RB := 1;
    LA := 7; -- limit for region A

```

```

  LB := 7; -- limit for region B
  loop
    select
      only if
        -- description of movement of train0
        (P0 < 6) and
        (T0 [P0+1] != T1 [P1]) and -
        (T0 [P0+1] != T2 [P2]) and -
        (T0 [P0+1] != T3 [P3]) and
        (T0 [P0+1] != T4 [P4]) and
        (T0 [P0+1] != T5 [P5]) and
        (T0 [P0+1] != T6 [P6]) and
        (T0 [P0+1] != T7 [P7]) and
        (RA + A0 [P0+1] <= LA) and
        (RB + B0 [P0+1] <= LB)
      then
        MOVE (0 of Train_Number);
        P0 := P0 + 1;
        RA := RA + A0 [P0];
        RB := RB + B0 [P0];
      end if
    [] only if
      . . .
    [] only if
      -- description of movement of train 7
      . . .
    then
      . . .
    end if
  [] only if
    -- condition for successful completion
    (P0==6) and (P1==6) and (P2==6) and (P3==6)
    and
    (P4==6) and (P5==6) and (P6==6) and (P7==6)
  then
    ARRIVED
  end if
end select
end loop
end var
end process

```

Verification The CADP framework allows to generate and export the whole state-space in standard formats (e.g. .bgc, .aut, .dot), to minimize the graph according to several equivalence relations, to display and edit its graphical layout, and to verify properties over it. The verification can be carried "on the fly", i.e. generating the fragment of the state-space actually used by the evaluation of a logical formula.

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.((( not ARRIVED]X) and (<true> true))
```

The evaluation of the above formula is completed in about 29 seconds.

As in the mCRL2 case, using regular expressions to denote transition sequences, the absence of deadlock can be checked with the formula `[true*]<true>true`. The other branching time formula – equivalent to `AG EF ARRIVED` can be expressed, using regular expressions, as:

```
[ true* ] < true* . ARRIVED > true
```

Model Variants Also in this case have have made an experiment in which the system was modelled a parallel composition set of interacting processes (one process for each itinerary endpoint, one process for each critical section and one process for each train). As in the mCRL2 case, the alternative modelling does not result more performing than the sequential case (evaluation time raises from 29 seconds to 15 minutes) and is therefore not further discussed.

10 The Round-trip Model

The system described in Section 2, where trains move only one way from one side the other side of the yard, has been chosen as the main reference for the illustration of the possible encodings of the blackboard style design into the seven selected frameworks. Indeed the main goal of our work is precisely to show the feasibility and the advantages resulting from this possibility of diversity, and a simple case study that does not require the exploitation of very specific tool features for being experimented has been considered as an appropriate choice for illustrating our idea.

We have however mentioned that our case study is a simplification of a more complex case study where trains cyclically perform never ending round missions along the yard. In this section we outline how this more complex case can be analysed in the various frameworks and the impact that this complexity has on the verification issues.

First of all we make an important observation. We are interested in proving that the critical section constraints that have been added at each movement step of

any train are sufficient to avoid the occurrence of deadlocks or livelocks, and therefore guarantee that each train has the possibility to always continue to advance – and reach its destination in presence of a fair dispatching policy. We can observe that any two trains of the same colour (i.e. T0-T4, T1-T5, T2-T6, T3-T7) always perform exactly the same cyclic mission, even if starting from different points. Therefore, the two system states in which two trains of the same color swap their identity are perfectly equivalent in terms of the overall system behavior. The exploitation of this symmetry leads to the consequence that the correctness of the system can be analysed by just observing the eight trains to perform a single round-trip mission, instead of an infinite sequence of them. This observation has a major impact on the system verifiability: not only it reduces the overall size of the problem¹⁵, but it removes the complexity of having to deal with livelocks issues because the presence of a blocked train will eventually lead to a complete system deadlock.

Given this premise, we can now show how the given initial case study can be easily extended to the case of single *round-trip* missions.

The first step is to extend the definition of the missions of the trains and the definitions of the tables of constraints governing the traversal of the A and B critical sections. In the case of UMC, we will have:

```
Vars:
-- mission steps for train T0
T0: int[] := [ 1, 9,10,13,15,20,23,
              22,17,18,11, 9, 2, 1];
. . .
T7: int[] := [26,22,17,18,12,27,
              8, 7,27,11,13,16,20,26];
. . .
-- region A constraints for train T0
A0: int[] := [ 0, 0, 0, 1, 0,-1, 0,
              1, 0, 0,-1, 0, 0, 0];
. . .
A7: int[] := [ 0, 1, 0,-1, 0, 0, 0,
              0, 0, 1,-1, 0, 0, 0];

B0: int[] := [ 0, 0, 0, 1, 0,-1, 0,
              1, 0, 0,-1, 0, 0, 0];
. . .
B7: int[] := [ 0, 0, 0,-1, 0, 0, 0,
              0, 0, 1,-1, 0, 1, 0];
```

The second step is the update of the rules governing the advancements of trains, by extending the limit on the train progresses from 6 to 13:

```
Behavior:
-- rule for advancement of train T0
s1 -> s1
{- [P0 < 13 &
  -- all the rest of the rule remains unchanged
  T0[P0+1] != T1[P1] &
  . . .
```

¹⁵ The continuously cycling model (syntactically a minimal variation of the round-trip one) has 159,374,352 states and 810,710,977 transitions.


```
}

```

Finally we have to update the detections of the final condition, when all the trains have completed their round mission:

```
Abstractions {
  State SYS.P0=13 and
    SYS.P1=13 and
      .
      .
      .
    SYS.P7=13 -> ARRIVED
  -- abstract label on final node
}
```

Mutatis mutandis, i.e. after performing the few syntactic changes that differentiate one encoding from the other, the same change can be repeated for all the frameworks taken into consideration.

This further analysis leads to the discovering of two novel cases of deadlocks (see Fig. 9), that require the definition of two other critical sections (that we call C and D), the definition of their corresponding constraints to be applied at each step by any trains, and the extension of the train advancement rules by taking into account also these new sections. For example, the rule governing the movement of train T0 will, in the end, become:

```
s1 -> s1
{- [P0 < 13 &
  T0[P0+1] != T1[P1] &
  T0[P0+1] != T2[P2] &
  T0[P0+1] != T3[P3] &
  T0[P0+1] != T4[P4] &
  T0[P0+1] != T5[P5] &
  T0[P0+1] != T6[P6] &
  T0[P0+1] != T7[P7] &
  RA + A0[P0+1] <= LA &
  RB + B0[P0+1] <= LB &
  RC + C0[P0+1] <= LC &
  RD + D0[P0+1] <= LD ] /
P0 := P0 + 1;
RA = RA + A0[P0];
RB = RB + B0[P0];
RC = RC + C0[P0];
RD = RD + D0[P0];
}
```

The formula to be verified will remain the same in the case of UMC, mCRL2, FDR4, CADP, and will have to take into account the new mission length in the case of SPIN and NuSMV.

This new single cycle model now appears to have 91,890,065 states, 453,321,793 transitions, and its verification time ranges from 34 minutes in the case of NuSMV, to 166 minutes in the case of mCRL2, and up to 16 hours in the case of UMC (see Table 3).

In the case of CADP, SPIN and UMC we have been forced to execute the verification not directly under the macOS environment but under a Linux virtual machine running within the macOS. This has been necessary, in the case of CADP to be able to use the 64-bit version of the tool (not yet available under macOS), and in the other two cases to be able to use an amount of virtual

memory greater than 1.5 times the size of the physical RAM of the computer. It happens in fact that the macOS kernel autonomously decides to kill the greatest memory eating application when the overall system response time would risk to be too damaged by the presence of memory hungry user applications. The best results in the case of UMC are obtained by adopting a depth first strategy, omitting the recording of counterexamples related informations, and using multiple cores during state space generation. The extremely high amount of memory used by UMC – which induces the extremely high evaluation time because of the amount of virtual memory swapping required – reflects the fact that the tool has not been designed with the goal of performing extremely large system validations, but rather with goal of facilitating the debugging of reasonably small system designs. The use of FDR4 requires the explicit setting of a disk based storage directory to avoid to hit the maximum amount of virtual memory allowed by the OS kernel. Any attempts to use SPIN under macOS always fails with process analyser being killed by the kernel. Under the Linux virtual machine, SPIN successfully verifies the model only forcing a breadth first evaluation strategy, requesting the use of on disk memory allocation, and by extending the default cache and vectors size. In the case of mCRL2, no particular measures have to be taken to tailor the use of virtual memory, while the `-vrjittyc` option should be used to get a satisfactory evaluation time. In the case of SMV and CADP the verification tools can be used in their default configuration without having to specify any particular evaluation choice. It is quite impressing the amount of virtual memory that appears to be used by NuSMV, i.e., only 1.4 GB.

11 Discussion

The pattern of having a set of global data that is concurrently and atomically updated by a set of concurrent guarded agents [21] is an architectural pattern often encountered in many fields. 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 formalized and verified using different languages and frameworks. We have experimented with seven possible alternatives, i.e., UMC, NuSMV, Promela/SPIN, mCRL2, FDR4, CPN and CADP, which differ greatly in expressive power, and support different verification approaches. The best results obtained, together with the options used for each framework, are described in more detail in the previous sections, and are summarized in Tables 2 (for the one-way, initial case) and 3 (for the round-trip case). The activity is still in progress, since, on the one hand, we plan to extend our experiments to several other well known toolsets, and, on the other hand, there are still many as-

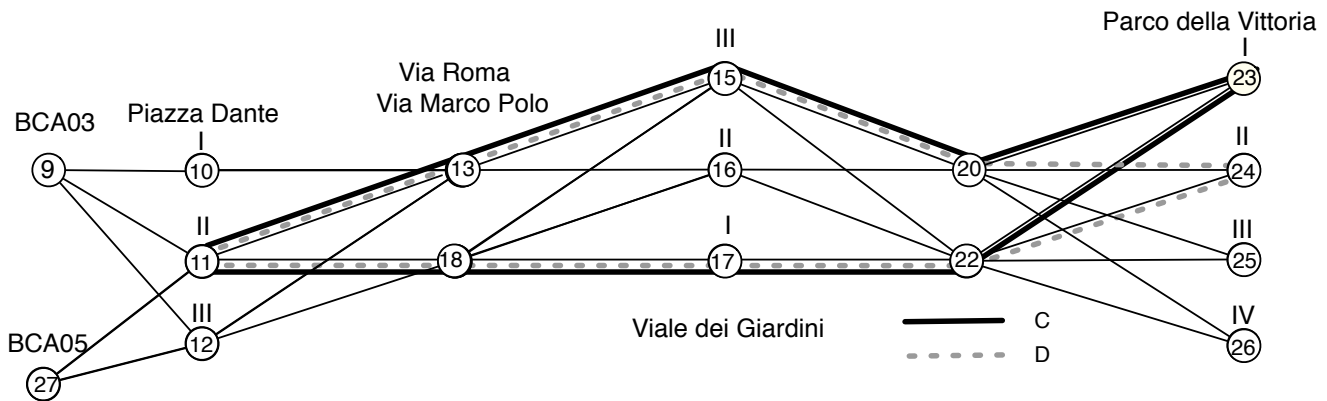


Fig. 9: The new critical sections C and D

pects 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 four broad parameters that had an impact on our experience, namely (1) specification formalism, (2) property definition language, (3) platforms compatibility, and (4) 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 and general conclusions cannot be drawn, we believe that it can offer a useful perspective on the applicability of the tools to specific problems of the railway context.

In the following paragraphs we describe the parameters, and, based on them, we compare the different tools, while in Table 1 and Fig. 10, we summarize our evaluation¹⁶.

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. In this paper, 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 supports an explicit shared data structure, seems the most intuitively suitable for the for-

malization of our problem, in which agents atomically read and update a common data blackboard. Process algebras are usually more oriented to model designs with communication agents that do not share a global status. Nevertheless, in our case, the algebraic model of the system, in which a system state is represented by process parameters (see Sect. 6 and 7), does not seem very distant from the other state-machine oriented representation. This is particularly evident in the case of CADP: the LNT specification has the aspect of a classical imperative state-machine oriented representation, while it is automatically transformed by the tool into a classical set of LOTOS algebraic processes.

With Petri Nets, the system state was concealed in the colour of a token (Sect. 8). In this case, our model with a single place is definitely 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 simple blackboard design solution, with the different tools. However, since other frameworks rely on a different kind of design approach with respect to the original state-machine oriented one – as mentioned, process algebras and Petri Nets –, we felt compelled to try see what would have happened if a different design were used (see Sect. 6, Sect. 7, Sect. 9 and Sect. 8). This has led to the observation that sometimes (e.g., in the case of FDR4) a different and more compositional design approach might actually induce much better performance, suggesting a possible solution to scalability problems of the verification effort. Another consideration is that the choice of using different design strategies might be interesting also from the point of exploiting *design diversity* – instead of tool diversity only – as another approach for the improvement of the overall trustworthiness of the verification process.

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. UMC

¹⁶ In Table 1 we show the time ranges for the one-way case. We do not show time ranges for the round-trip case, since these times are highly influenced by the memory swapping, and different operating systems were used for the round-trip experiments, due to the constraints explained in Sect. 10

instead 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). It is not a surprise that this additional complexity takes its toll in terms of performance.

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. Our initial property, that all execution paths end in certain state, is a very simple property which can easily be checked in all frameworks, either as a CTL formula, or as a LTL formula, or a CSP specification to be refined. However, it is useful to briefly summarize 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. For example, our generalized case study involving continuously cycling trains (see Sect. 3) might give raise to relevant verification difficulties in frameworks that do not support truly branching time formulas. The most powerful environments in terms of property definition language are mCRL2 and CADP (both event based), that support a parametric version of μ -Calculus (that subsumes both LTL and CTL). Also, UMC supports plain μ -Calculus, even if in its plain non-parametric form. The original point of UMC is that it support both state and event based approaches, allowing to write formulas that can take into account both predicates over the states and conditions over of the events occurring during a system evolution. The property specification language supported by SPIN is the classical (state based) Linear Time Logics (LTL), and we have seen that this choice might lead to difficulties in specifying and verifying livelock related properties in the case of cyclic models. NuSMV instead supports directly both LTL and CTL (in their state based versions). 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. This approach has its own advantages (e.g. in terms of compositionality) and disadvantages (e.g. in the difficulty of finding the correct specification). In our case we we have observed that we might have difficulties in specifying and verifying livelock related properties (see Sect. 7).

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 software 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. While

most of our experiments were performed on directly under macOS, in case of CPN tools a Windows emulator was used. While in the cases of UMC and SPIN a Linux emulator has instead been used (for the more complex case studies) to overcome the limits of our native OS, in the case CADP it has been used to exploit a more advanced version of the tool – see Sect. 10.

Performance Fig. 10 summarises the execution time ranges observed in our experiments for the simpler one-way case study. Each point in a range corresponds to the use of a specific evaluation option or to a specific variation in the system design. The actual code and evaluation instructions for each specific case can be found in our data repository [59]. It was outside our goals to make a rigorous comparative evaluation of the performance of the various approaches, and the data shown here should be considered as indications of the observed experiments. For example, we did not try to exploit the “swarm” feature of SPIN (taking advantage of multicore architectures) or of the “cluster” features of CADP (taking advantage of distributed architectures). Almost all the tools show extremely great differences in terms of evaluation times depending on the design or evaluation choices done by the user. This fact seems to indicate that a deep mastering of the tools is required to exploit at their best the capabilities of the various frameworks, and this fact is somewhat an obstacle for our goals of applying diversity in tool selection. It might be expensive to become really experts in many different frameworks. In particular, process algebraic (FDR4, mCRL2) approaches appeared to be very sensitive to design variations (e.g. sequential versus parallel designs, alternative ways of composing parallel processes), while SPIN and mCRL2 are probably the most sensitive frameworks from the point of view of user options applicable at verification time.

12 Towards Formal Methods Diversity

The possibility to model and verify a certain design with completely different verification frameworks can be an interesting solution from the point of view of the *validation* of critical systems. The CENELEC EN 50128 norms [13], for the development of railway software, asks the tools used along the process to be qualified, or certified, for their usage in the context of safety-critical products development. With some limited exceptions, i.e., SCADE [22], none of the verification tools available, including the ones considered in this study, is designed and validated at the greatest safety integrity levels by itself. However, 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 engineer-

Table 1: Summary of the different environments

Tool	Version / Year	Specification Language	Property Definition Language	Platform Compatibility
UMC	v. 4.6 / 2017	State Machines (Structured Data)	μ -Calculus CTL/ACTL	Online, Unix, Windows, macOS
NuSMV	v. 2.6.0 / 2015	State Machines (Flat Data)	CTL/LTL/PSL	Unix, Windows, macOS
SPIN	v. 6.4.7 / 2017	State Machines (Flat Data)	LTL	Unix, Windows, macOS
mCRL2	v. 201707.1 / 2017	Process Algebra (Algebraic Data)	parametric μ -Calculus	Unix, Windows, macOS
CPN Tools	v. 4.0.1 / 2015	Petri Nets (Functional Data)	ML functions CTL	Windows
FDR4	v. 4.2.3 / 2017	Process Algebra (Functional Data)	Refinement Checking	Unix, Windows, macOS
CADP	v. 2017-j / 2017	LNT/LOTOS (Structured Data)	MCL	Unix, Windows, macOS

Table 2: Data summary for the one-way experiments – best cases only.

Tool	System	Num. States	Best Encoding Structure	Best Evaluation Time	VMem.	Commands
UMC	macOS	1,636,545	sequential	38 secs	2.9 GB	umc -m3 -100 umc_oneway8.txt AF.txt
NuSMV	macOS	1.63654e+06	sequential	2.9 secs	74 MB	nusmv -r -v 1 smv_oneway8.smv
SPIN	macOS.	1,636,546	sequential	13 secs.	1 GB	time spin -a spin_oneway8.pml; gcc -O3 -o pan pan.c; pan -v
mCRL2	macOS	1,636,545	sequential	113 secs.	1 GB	mcr122lps mcr12_oneway8.txt temp.lps; lps2pbes -fmuAF.mcf temp.lps temp.pbes; pbes2bool -s2 -vrjittyc temp.pbes;
CPN	n/a	n/a	n/a	n/a	n/a	n/a
FDR4	macOS	1,636,546	parallel	41 secs	650 MB	refines fdr4_oneway8.txt
CADP	macOS	1,636,545	sequential	29 secs.	78 MB	lnt.open cadp_oneway8.lnt evaluator4 AF.mcl

Table 3: Data summary for the round-trip experiments – best cases only.

Tool	System	Num. States	Best Encoding Structure	Best Evaluation Time	VMem.	Commands
UMC	Linux	91,890,065	sequential	16 hours	112 GB	umc -m3 -110 umc_round8.txt AF.txt
NuSMV	macOS	9.18901e+07	sequential	34 min.	1.4 GB	nusmv -r -v 1 smv_round8.smv
SPIN	Linux.	91,890,066	sequential	185 min.	102 GB	time spin -a spin_round8.pml; gcc -O3 -DBFS -DBFS_DISK -DVECTORSZ=256000 -o pan pan.c; pan -v
mCRL2	macOS	91,890,065	sequential	145 min.	43 GB	mcr122lps mcr12_round8.txt temp.lps; lps2pbes -fmuAF.mcf temp.lps temp.pbes; pbes2bool -s2 -vrjittyc temp.pbes
CPN	n/a	n/a	n/a	n/a	n/a	n/a
FDR4	macOS	91,890,066	parallel	60 min.	31 GB	refines --refinement-storage-file-path swapdir fdr4_round8.txt
CADP	Linux	91,890,065	sequential	79 min.	7.3 GB	lnt.open cadp_round8.lnt evaluator4 AF.mcl

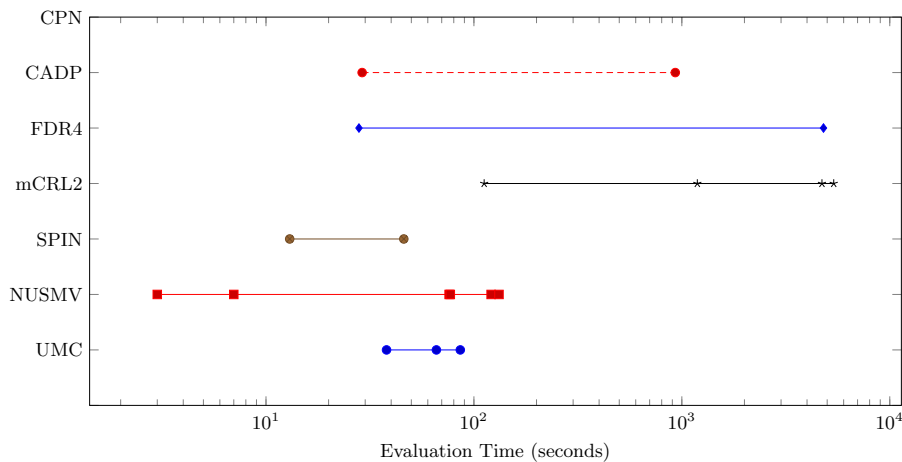


Fig. 10: Summary of evaluation time ranges - one-way case (logarithmic scale).

ing [52,62], in the application of formal methods. More specifically, we suggest to use different non-certified formal environments for the modelling and verification of a certain 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*¹⁷ of an algorithm, by ensuring that the encoding of the specification into 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 [60]. 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. In Sect. 2 we have already described that our abstract system design has a precise number of states and transitions, that corresponds to the train positions and the allowed train movements. If we look at the data of Table 2 and Ta-

ble 3 we can indeed verify the precise size of this state space in the one-way (1,636,535 states) and round-trip cases (91,890,065 states). The fact that all the encodings report the same size¹⁸ is an encouraging indicator on the correctness of the translation. If the number of states is the same, and all the specifications satisfy the same properties, this increases the confidence on the equivalence of the specifications. Further validation of the translation (see, e.g., [3]) is however required to ensure that the specification verified is equivalent for the different environments.

It is worth noticing that this does not fully guarantee that the specification itself is free from faults, since initial faults may be propagated from the original specification. To achieve higher confidence on the results, one shall also pursue requirements validation, as explained later in this section.

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 known, the two logics are not comparable [17], 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. It is also worth noticing that the encoding of the property to be verified can be a possible source of error. When

¹⁷ The concept of specification is intended here in Jackson's terms [40], i.e., the model that, given certain environmental assumptions, shall satisfy the requirements.

¹⁸ a difference of +1 or +2 among the models is due to the different way in which is modeled the system initialization and the system final state

writing a LTL, CTL, MCL, μ -calculus formula or an algebraic specification, it is definitely not difficult to make mistakes. Also in these cases a comparison of the results of the verification might help in identifying and reducing this source of errors. 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.* [32] in the context of information systems verification, would 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 [14], 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 [6, 52]. 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 lose the benefits of modelling diversity. On the other hand, if also code diversity is employed [6], 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 [27]. Rigorously defining a railway process, adherent to the CENELEC EN 50128 norm [13], and based on formal methods diversity is beyond the scope of this paper. However, a rigorous definition 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, since the *performance* of the tools is affected by (a) design decisions, as we have shown, e.g., for FDR4

(see Sect. 7) and (b) verification options, as shown by the different time ranges obtained in our experiments, reported in Fig. 10. Therefore, if one is oriented to exploit the capabilities of different tools at their best, high proficiency is required with different tools. 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 [31].

Appropriateness of a Formal Tool for a Design We have seen that our algorithm design can be represented with seven 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 [72]. For example, in the literature we see that state-based graphical specifications are used to model the control logic of ATC/ATP systems [28, 47, 63, 15, 27], while interlocking systems are often modelled with textual specification, and verified by means of model checking [70, 68, 44, 29, 11, 49]. 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 [72] in the context of network protocols, shall be performed in the railway domain to achieve this goal. The authors of this paper are currently exploring this issue in the framework of the ASTRail European Project¹⁹, and results on this aspect will be provided in future publications.

Evolution and Acceptance of Formal Tools The tools that we used in our experiments are freely available (sometimes with an academic licence), and mostly maintained by universities or public organizations. Even within the time span in which this paper was written, evolution in terms of versions of the tools was observed (e.g., almost all the tools have had at least a new version in 2017). 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 [28], railway companies are keen to prefer commercial tools especially for the availability of assistance and the support of legacy versions. We are aware that, in general, also the open source world is

¹⁹ <http://www.astrail.eu>

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.

13 Conclusion

The world of formal methods offers several options in terms of automated environments [32, 23], which can and have been used to verify the design of railway systems [24, 36]. In this paper, we show the application of seven different formal tools, namely UMC, Promela/SPIN, NuSMV, mCRL2, CPN Tools, FDR4 and CADP, in the modelling and verification of a deadlock avoidance algorithm for train scheduling [57]. 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 [21], in which concurrent guarded agents atomically update a global data blackboard. Our experience shows that small choices in the design or verification options trigger radical changes in terms of performance, especially for process algebraic (FDR4, mCRL2) approaches. Furthermore, we have observed 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 a replication of the same design allows formal methods users 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. Specific challenges include, for example: the need to have diverse formal methods expertise available within railway companies; the preference of these companies towards – possibly stable – commercial tools, in a context in which formal tools are often open source, academic platforms; the need to restructure the development process to accommodate formal methods diversity, while keeping the process compliant to the CENELEC norms [13]. To fully establish, and possibly automate, the idea, approaches for model transformation, and for the verification of the correctness of the translation step [3], should also be applied.

Our personal interest is now to further experiment with additional free and open source tools, such as LTS-Min [10], LTSA [53], DiVinE [7], JavaPathFinder [43], Alloy [48], and commercial tools, such as SCADE [22],

and Stateflow with Simulink Design Verifier [42]. Our idea is to model our prototypical railway problem, as well as other cases, 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. Furthermore, we are also interested in comparing the different tools in terms of clarity of the counterexamples provided, and in terms of effectiveness in discovering specific types of errors in the specifications. In this sense, we wish to investigate also the potential *complementarity* of the different tools, in a development context in which formal methods are diversified, as proposed in this paper. It should finally be noticed that the presented reflections, especially those concerning the different tools, come from an experience report, with the limits entailed by this research approach. Although general conclusion cannot be drawn, and even though the reflections provided are driven by the authors' experience, we believe that our work can offer inspiration for future research in the field.

Acknowledgments

This work has been partially funded by the ASTRail project. This project received funding from the Shift2Rail Joint Undertaking under the European Union's Horizon 2020 research and innovation programme under grant agreement No 777561. The content of this paper reflects only the authors' view and the Shift2Rail Joint Undertaking is not responsible for any use that may be made of the included information.

References

1. Accellera, property specification language - reference manual - version 1.01. <http://www.eda.org/vfv/docs/psl/lrm-1.01.pdf>, 2003.
2. Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.
3. Moussa Amrani, Levi Lucio, Gehan Selim, Benoît Combemale, Jurgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R Cordy. A tridimensional approach for studying the formal verification of model transformations. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 921–928. IEEE, 2012.
4. Marc Antoni and Nadia Ammad. Formal validation method and tools for french computerized railway interlocking systems. *IET Conference Proceedings - 4th IET International Conference on Railway Condition Monitoring (RCM 2008)*, pages 6–6(10), 2008.
5. André Arnold, Marie Claude Gaudel, and Bruno Marre. An experiment on the validation of a specification by heterogeneous formal means: The transit node. In *5th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA5)*, pages 24–34, 1995.

6. Algirdas Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.
7. 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.
8. 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.
9. Nazim Benaissa, David Bonvoisin, Abderrahmane Feliachi, and Julien Ordioni. The perf approach for formal verification. In *International Conference on Reliability, Safety and Security of Railway Systems*, pages 203–214. Springer, 2016.
10. 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.
11. 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.
12. Susan S. Brilliant, John C. Knight, and Nancy G. Leveson. Analysis of faults in an n-version software experiment. *IEEE Transactions on software engineering*, 16(2):238–247, 1990.
13. CENELEC. EN 50128:2011: Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. Technical report, 2011.
14. 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.
15. 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.
16. 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.
17. Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.
18. 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.
19. Rocco De Nicola and Matthew CB Hennessy. Testing equivalences for processes. *Theoretical computer science*, 34(1-2):83–133, 1984.
20. Rocco De Nicola and Frits Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM)*, 42(2):458–487, 1995.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. Alessio Ferrari, Alessandro Fantechi, and Stefania Gnesi. Lessons learnt from the adoption of formal model-based development. *NASA Formal Methods*, pages 24–38, 2012.
27. 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.
28. Alessio Ferrari, Alessandro Fantechi, Gianluca Magnani, Daniele Grasso, and Matteo Tempestini. The métro case study. *Science of Computer Programming*, 78(7):828–842, 2013.
29. Alessio Ferrari, Gianluca Magnani, Daniele Grasso, and Alessandro Fantechi. Model checking interlocking control tables. In *FORMS/FORMAT 2010*, pages 107–115. Springer, 2011.
30. 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.
31. 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.
32. 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.
33. Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *STTT*, 15(2):89–107, 2013.

34. Hubert Garavel, Frédéric Lang, and Wendelin Serwe. From LOTOS to LNT. In *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 3–26. Springer, 2017.
35. 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.
36. Stefania Gnesi and Tiziana Margaria. *Formal methods for industrial critical systems: A survey of applications*. John Wiley & Sons, 2012.
37. 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.
38. Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and analysis of communicating systems*. 2014.
39. 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.
40. 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.
41. 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.
42. 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.
43. 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.
44. 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.
45. Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.
46. Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.
47. 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.
48. Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.
49. 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.
50. 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.
51. Kurt Jensen and Lars M Kristensen. *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media, 2009.
52. 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.
53. 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.
54. Radu Mateescu and Damien Thivolle. A model checking language for concurrent value-passing systems. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 2008.
55. 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.
56. 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.
57. 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.
58. 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.
59. Franco Mazzanti, Giorgio Oronzo Spagnolo, and Alessio Ferrari. Formal Tool Diversity – Experiments Data Repository. <https://github.com/ISTI-FMT/TrainSchedulingModels>, 2017.
60. Steven P Miller, Michael W Whalen, and Darren D Cofer. Software model checking takes off. *Communications of the ACM*, 53(2):58–64, 2010.
61. Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.
62. 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.

63. 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.
64. RTCA. DO-178C Software Considerations in Airborne Systems and Equipment Certification, 2012.
65. 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.
66. 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.
67. 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.
68. 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.
69. Michael Whalen, Darren Cofer, Steven Miller, Bruce H Krogh, and Walter Storm. Integration of formal analysis into a model-based software development process. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 68–84. Springer, 2007.
70. 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.
71. Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):19, 2009.
72. Pamela Zave. A practical comparison of alloy and spin. *Formal Aspects of Computing*, 27(2):239, 2015.