Experiments in Formal Modelling of a Deadlock Avoidance Algorithm for a CBTC System

Franco Mazzanti, Alessio Ferrari, and Giorgio O. Spagnolo

Istituto di Scienza e Tecnologie dell'Informazione "A.Faedo",
Consiglio Nazionale delle Ricerche, ISTI-CNR, Pisa, Italy

Abstract. This paper presents a set of experiments in formal modelling and verification of a deadlock avoidance algorithm of an Automatic Train Supervision System (ATS). The algorithm is modelled and verified using four formal environment, namely UMC, Promela/SPIN, NuSMV, and mCRL2. The experience gained in this multiple modelling/verification experiments is described. We show that the algorithm design, structured as a set of concurrent activities cooperating through a shared memory, can be replicated in all the formal frameworks taken into consideration with relative effort. In addition, we highlight specific peculiarities of the various tools and languages, which emerged along our experience.

Keywords: model checking, formal design, NuSMV, SPIN, UMC, mCRL2, comparison of model checkers, CBTC, deadlock avoidance, railways.

1 Introduction

In this paper, we show that a representative railway problem can be modelled and verified with limited effort using four different tools, namely: UMC, Promela/SPIN, NuSMV, and mCRL2. 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 [8, 9] of a Communications-based Train Control System (CBTC) [17]. 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 – which can be modelled as a set of global data that is concurrently and atomically updated by a set of concurrent guarded agents – i.e., agents that, when certain global conditions are met, are allowed to atomically change the global status. In the paper, we show relevant excerpts of the models and the verification results.

From our experience, we saw that small choices in the specification of the models, or in the verification options, can greatly impact on the verification

time. With some differences, this observation holds for all the modelling frameworks considered. Hence, we argue that a deep proficiency with each one of the frameworks is required to effectively exploit their verification capabilities.

The rest of the paper is structured as follows. In Sect. 2 we describe the deadlock avoidance algorithm that we modelled. In Sect. 3–6, we show our models and the verification results for UMC, NuSMV, Promela/SPIN and mCRL2, respectively[1], and, within the descriptions of the models, we highlight the peculiarities of the different languages and environments. Finally, Sect. 7 concludes the paper and provides general observations on the experience.

2 The Deadlock Avoidance Algorithm

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

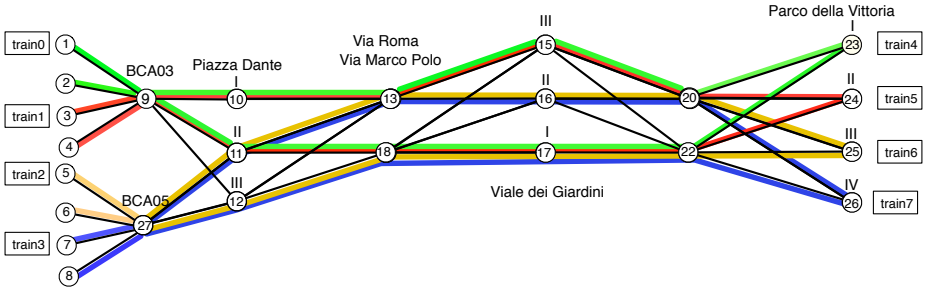

Fig. 1: A fragment of the yard layout and the 8 missions of the trains

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

[1] All the verification experiments have been conducted on a Mac Pro (late 2013) workstation with *Quad-core 3,7Ghz Intel Xeon E5, 64 GB RAM* running OS X 10.11 (El Capitan). All the models referred in this paper can be retrieved from the URL http://fmt.isti.cnr.it/WEBPAPER/ISOLA2016data.zip

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 [8, 9]. 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 $i = 0 \ldots 7$, 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.

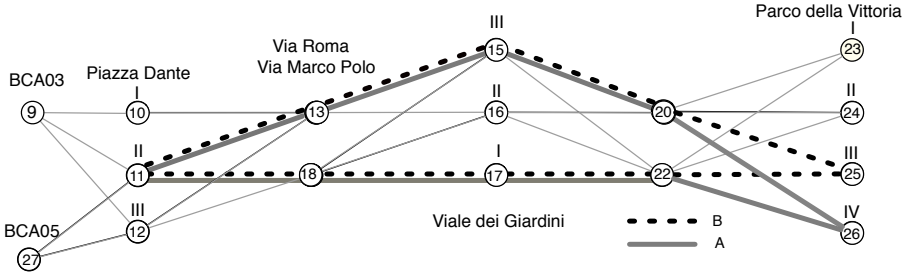

Fig. 2: The critical section A and B which must not be saturated by 8 trains

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

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

3 The UMC Model

UMC [4] is a model checker that belongs to the KandISTI [2, 3] family. Its development started at ISTI in 2003 and has been since then used in several research projects. So far UMC is not really an industrial scale project but more

an (open source) experimental research framework. It is actively maintained and is publicly usable through its web interface [5].

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 [7]. Labels on the states represent basic state properties, and labels on the edges represent properties of system transitions. The logic supported by the KandISTI framework uses the evolution graph as semantic model and allows to specify abstract properties in a way that is rather independent from the internal implementation details of the system [6].

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

The `Vars` part contains the vectors describing the train missions (T_i), the indexes recording the train progresses (P_i) – i.e., the indexes in the previous vectors –, the occupancy counters RA and RB of the two 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:
  T0: int[] := [ 1, 9,10,13,15,20,23]; --  mission steps for train0
    . . .
  T7: int[] := [26,22,17,18,12,27, 8]; --  mission steps for train7
  LA: int :=7;                         -- limit value for region RA
  A0: int[] := [0, 0, 0, 1, 0,-1, 0];  -- RA updates steps for train0
    . . .
  A7: int[] := [1, 0, 0, 0,-1, 0, 0];  -- RA updates steps for train7
  RA: int :=1;                         -- occupancy of region RA
  LB: int :=7;                         -- limit value for region RB
  B0: int[] := [0, 0, 0, 1, 0,-1, 0];  -- RB updates steps for train0
    . . .
  B7: int[] := [1, 0, 0, 0,-1, 0, 0];  -- RB updates steps for train7
  RB: int :=1;                         -- occupancy of region RA
  P0,P1,P2,...,P7:int :=0;             -- train progresses
```

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 un-

bounded) events queues can contribute to make the actual size of a state highly dynamic. This dynamism might lead to potentially infinite state systems.

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

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

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

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

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

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

As a last step we have to define what we want to see on the abstract L2TS associated to the system evolutions. Indeed, the overall behaviour of a system is formalised as an abstract doubly labelled transition system (L2TS), and abstraction rules allow to define what we want to see as labels of the states and edges of the L2TS. The abstraction rules are expressed in the `Abstraction` part of the specification, in which we define which labels should appear on the edges and states of the abstract evolution graph. In our case, we are interested to observe the existence of a certain state where all trains have completed all their missions. This can be done assigning a state label, e.g. `ARRIVED`, to all the system configurations in which each train is in its final position.

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

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

```
AF ARRIVED
```

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

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

In our case the formula is *true* and UMC completes the evaluation in a time which ranges from 28 seconds to 106 seconds depending on how the tool is used. The fastest results of 28 seconds is obtained by exploiting a prototypal parallel version of UMC ([10]), by adopting a depth-first exploration strategy, and letting the evaluation to proceed in a non-interactive way which does not collect the data necessary for a subsequent explanation of the results.

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

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

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

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

At this point the property to be verified becomes:

```
AF {arrived} true
```

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

4 The NuSMV Model

NuSMV [13, 14] is a software tool for the formal verification of finite state systems. NuSMV was jointly developed by FBK-IRST and by Carnegie Mellon University. NuSMV allows to check finite state systems against specifications in the Computation Tree Logic (CTL), Linear Temporal Logic (LTL) and in the Property Specification Language (PSL)[1].

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

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

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

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

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

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

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

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

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

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

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

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

```
      : RA + A0[P0+1]; -- RA updated according to movement of train0
       --
      RUNNING =1 &          -- train1 selected for possible evolution
       . . .                 -- train1 actually allowed to move
      : RA + A1[P1+1];   -- RA updated according to movement of train1
       . . .
      TRUE    -- no train can move (deadlock or all trains arrived)
      : RA;   -- RA remains the same
    esac;
```

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

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

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

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

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

```
FAIRNESS  RUNNING = 0;
   . . .
FAIRNESS  RUNNING = 7;
```

In this way, NuSMV limits its evaluations to the fair paths of the system evolutions, i.e. those infinite paths for which the fairness constraints are true for an infinite number of times. With the above constraints, an infinite path in

which only `train0` is selected is discarded, because it violates the fairness rules `RUNNING=1,...,RUNNING=7`.

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

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

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

5 The Promela/SPIN Model

SPIN [11, 12] is an advanced and very efficient tool specifically targeted for the verification of multi-threaded software. The tool was developed at Bell Labs in the Unix group of the Computing Sciences Research Center, starting in 1980. In April 2002 the tool was awarded the ACM System Software Award. The language supported for the system specification is called Promela (PROcess MEta LAnguage). Promela is a non-deterministic language, loosely based on Dijkstra's guarded command language notation, and borrowing the notation for I/O operations from Hoare's CSP language. Once a model is formalised in Promela, a corresponding analyser in generated as a source C program (`pan.c`). The compilation and execution of the analyser performs all the needed on-the-fly state generations and verification steps. The properties to be verified can be

expressed in LTL, and a violation of a property can be explained by observing the generated counterexample trail path.

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

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

```
byte  T0[7];       //  mission data for train0
 . . .
byte  T7[7];      //  mission data for train7
byte  P0,...,P7; //  progress data for train0,...train7
byte  RA;         // occupancy of region A
byte  RB;         // occupancy of region B
byte  LA;         // limit of region A
byte  LB;         // limit if region B
short A0[7];      // increments/decrements of train 0 for Region A
 . . .
short A7[7];      // increments/decrements of train 1 for Region B
short B0[7];      // increments/decrements of train 0 for Region B
 . . .
short B7[7];      // increments/decrements of train 7 for Region B
```

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

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

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

The system initialisation/execution code (c) consists of: 1) the setting of the initial value for the state variables; 2) the possible activation of concurrent, communicating, asynchronous subprocesses (sharing the same global memory); the main execution of a sequence of statements. In Promela, sequences of statements, when included inside an `atomic {...}` construct, are executed as part of a single system (or process) transition.

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

```
init {
  atomic {   //  initializations of state variable
   // T0:[1,9,10,13,15,20,23]
   T0[0]=1;  T0[1]=9;  T0[2]=10; T0[3]=13; T0[4]=15;
   T0[5]=20; T0[6]=23;
    . . .
   // T7:[26,22,17,18,12,27,8]
```

```
  T7[0]=26; T7[1]=22; T7[2]=17; T7[3]=18; T7[4]=12;
  T7[5]=27; T7[6]=8;
  A0[3]= 1; A0[5]= -1;              // A0:[0,0,0,1,0,-1,0]
   . . .
  A7[1]=1;  A7[4]=-1;               // A7:[0,1,0,0,-1,0,0]
  B0[3]=1;  B0[5]=-1;               // B0:[0,0,0,1,0,-1,0]
   . . .
  B7[0]=1;  B7[4]=-1;               // B7:[1,0,0,0,-1,0,0]
  RA=1;     RB=1;     LA=7;     LB=7;
}       . . .// end of initializations of state variables
  . . . // activation of subprocesses
  . . . // main sequence of statement
```

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

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

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

The evaluation of the formula is carried over by the process analyser (`pan.c`) in about 25 seconds, which decrease to 10 seconds when the process analyser is compiled with all `gcc` optimisations turned on (-O3 flag). We have also experimented the version of this specification in which each train was represented by an explicit process, whose activity consists in just executing the loop of its own

atomic progress transition. This architecture, indeed, is the one which more precisely reflects our logical system design. In this case, the evaluation time raises to about 126 seconds (which decrease to about 47 seconds with `gcc` optimisations turned on).

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

6 The mCRL2 model

mCRL2[15, 16] is a formal specification language with an associated toolset. The toolset can be used for modelling, validation and verification of concurrent systems and protocols. The mCRL2 toolset is developed at the department of Mathematics and Computer Science of the Technische Universiteit Eindhoven, in collaboration with LaQuSo, CWI and the University of Twente. The mCRL2 language is based on the Algebra of Communicating Processes (ACP) which is extended to include data and time. Processes can perform actions and can be composed to form new processes using algebraic operators. A system usually consists of several processes, or components, running in parallel. A process can carry data as its parameters. The state of a process is a specific combination of parameter values. In our case, we need to model the existence of a global status shared among the various trains, and this can be represented in mCRL2 by a single, recursive, non-deterministic process, whose parameters precisely model the global system state. Also in this case, the non-determinism of the system evolutions is modelled through the non-determinism of the main process behaviour.

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

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

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

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

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

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

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

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

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

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

   +   % all trains have completed their missions
   ((P0 ==6) && (P1 ==6) && (P2 ==6) && (P3 ==6)  &&
    (P4 ==6) && (P5 ==6) && (P6 ==6) && (P7 ==6)
    ) ->
     arrived . AllTrains(P0,P1,P2,P3,P4,P5,P6,P7,RA,RB);
```

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

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

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

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

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

The above formula is just a translation in μ-calculus of action-based CTL-like formula **AF** {`arrived`} `true` used with UMC. We refer to [6] and [15] for detailed description of the semantics of these two logics.

The evaluation of this formula takes from 1 to about 19 minutes before returning the *true* value, depending on the options selected during the various evaluation steps. The greatest impact, which reduces the evaluation time from 19 minutes to about 1 minute and 40 seconds, is obtained with the selection of the `jittyc` data rewriting mode.

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

7 Discussion and Conclusion

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

The activity is still in progress, since, on the one hand, we plan to extend our experiments to several other well known toolsets, and, from the other hand, there are still many aspects of the currently explored four frameworks that need a deeper understanding and evaluation.

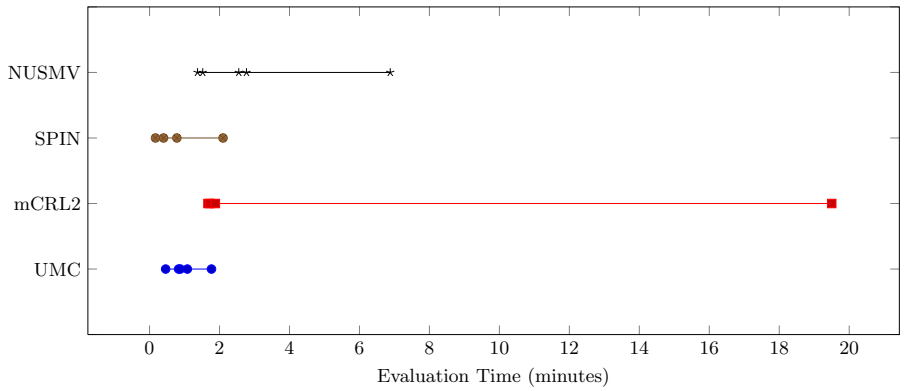

Fig. 3: Summary of evaluation time ranges for the 4 frameworks

Fig. 3 summarises the execution time for each model configuration adopted in our experiments[2]. Already from the data that we have collected we can observe that apparently small choices in the construction of the models, or in the selection of the best options for the evaluations, can greatly affect the performance of the verification. Almost all the tools show extremely great differences in terms of execution times depending on the choices done by the user. In our case, we obtained the best performances from NuSMV by declaring the RUNNING variable as **IVAR**. For SPIN, the usage of the -O3 flag (i.e., all `gcc` optimisations turned on) was the factor determining the major decrease in terms of verification time. For mCRL2, the selection of the `jittyc` data rewriting mode was crucial in increasing the performance. Finally, with UMC, the lower verification time was obtained with a parallel version of the tool, and selecting the non-interactive evaluation mode. The differences in terms of verification times obtained, and the different solutions adopted for each tool to minimise the verification time, indicate that a deep mastering of the tools is required to exploit at their best the capabilities of the various frameworks.

Another observation arising from our experiment, is that almost all platforms seem to be mainly tailored to the (successful) validation of a (correct) system design. When it comes to providing the user with an easy-to-understand description of why a given system design does not behave as expected, almost all the tools show great losses in terms of usability. One of the driving long term goals

[2] Each configuration corresponding to a point in Fig. 3, with associated verification time, is available at http://fmt.isti.cnr.it/WEBPAPER/ISOLA2016data.zip.

of the UMC/KandISTI project is precisely that one of supporting the user in a friendly way during the early steps of the system design, when the ideas might not yet be perfectly clear and the generated models still contain errors. The model checkers of the KandISTI framework are still far from a successful solution to this problem, however they seems to be moving in the correct direction towards a more easily usable early-designs verification environment. mCRL2 is probably the framework which currently suffers most from this point of view, probably because it is also the most powerful framework in terms of specification language and verification logic, and this makes particularly hard the construction of a user friendly explanation of the link between the property evaluation results and the operational system behaviour.

A final consideration which has been stimulated by our experimentation is that modelling and verifying a system using different approaches can really give a plus in the reliability of the verification results. We have actually experienced that the effort of modelling and checking a system design in several variants is essential for identifying the errors introduced in the construction of the formal specification or in the verification process. The possibility to model and verify a certain design with completely different verification frameworks can be an interesting solution also from the point of view of the *validation* of critical systems. Indeed, while none of the verification tools considered is designed and validated at the greatest safety integrity levels by itself, the existence of different, non validated, tools producing the same result might increase the overall confidence on the verification results.

References

1. Accellera, Property Specification Language - Reference Manual - Version 1.01. http://www.eda.org/vfv/docs/psl lrm-1.01.pdf, April 2003.
2. ter Beek, M., Gnesi., Mazzanti, F.: From EU projects to a family of model checkers - From Kandinsky to KandISTI. In: Software, Services, and Systems. LNCS, vol. 8950, pp. 312–328, Springer, Heidelberg (2015)
3. http://fmt.isti.cnr.it/kandisti
4. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Science of Computer Programming, Volume 76, Issue 2, pp. 119–135, Elsevier (2011)
5. UMC home site: http://fmt.isti.cnr.it/umc
6. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A logical verification methodology for service-oriented computing. ACM Transactions on Software Engineering and Methodology, Volume 21, Num. 3, ACM Press, (2012)
7. Gnesi, S., Mazzanti, F.: An Abstract, on the Fly Framework for the Verification of Service-Oriented Systems. In: Rigorous Software Engineering for Service-Oriented Systems, LNCS, vol. 6582, pp. 390–407, Springer, Heidelberg (2011)
8. Mazzanti, F., Spagnolo, G.O., Della Longa, S., Ferrari, A.: Deadlock avoidance in train scheduling: A model checking approach. In: Lang, F:, Flammini, F. (eds) Formal Methods for Industrial Critical Systems, FMICS 2014; LNCS, vol. 8718, pp. 109–123, Springer, Heidelberg (2014)

9. Mazzanti, F., Spagnolo, G.O., Ferrari, A.: Design of a deadlock-free train scheduler: a model checking approach. In: Julia M. Badger, J. M.0, Rozier, K. Y. (eds), NASA Formal Methods, NFM 2014. LNCS, vol. 8430, pp. 264–269, Springer, Heidelberg (2014)
10. Mazzanti, F.: An Experience in Ada Multicore Programming: Parallelisation of a Model Checking Engine. In: Reliable Software Technologies - Ada-Europe 2016. LNCS, vol.9695, pp. 94-109,Springer, Heidelberg (2016)
11. Holzmann,G.H.: The SPIN model checker. Addison-Wesley Pearson Education ISBN 0-321-22862-6, 2003
12. Verifying Multi-threaded Software with Spin. http://spinroot.com
13. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In Proceedings of Computer Aided Verification (CAV 02), (2002)
14. NuSMV: a new symbolic model checker. http://nusmv.fbk.eu/
15. Groote, J. F., Mousavi, M. R.: Modeling and Analysis of Communicating Systems. MIT Press, ISBN: 9780262027717 (2014)
16. MCRL2 analysing system behavior http://www.mcrl2.org/
17. Ferrari, A., Spagnolo, G. O., Martelli, G., Menabeni, S.: From commercial documents to system requirements: an approach for the engineering of novel CBTC solutions. STTT 16(6): 647-667 (2014)