

# Application of Model-Checking Techniques to Railway Scheduling Problems

Radu Mateescu, Wendelin Serwe, and Aline Uwimbabazi

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP\*, LIG, 38000 Grenoble, France  
`{first-name,last-name}@inria.fr`

**Abstract.** Railway scheduling problems can be formulated as instances of the NP-hard job-shop scheduling problem. In this paper, written in the honour of Alessandro Fantechi, we experiment with various encodings of the job-shop scheduling problem in LNT, a modern formal modelling language with a process calculus flavour. We also report on the computation of solutions with various model checking tools of the CADP toolbox.

## 1 Introduction

Railway transportation is a complex system of systems, encompassing a variety of control systems (interlocking, traffic management, automatic train control and supervision, etc.) that interact and operate together to ensure safety, availability, and performance of the transport. Railway control systems are developed according to international standards, such as ERTMS/ETCS<sup>1</sup> (European Rail Traffic Management Systems/European Train Control System) and feature embedded, cyber-physical, and distributed heterogeneous architectures. These systems are subject to high safety and availability requirements, which become even more stringent for the advanced (ERTMS/ECTS level 3) train control systems relying on geographically distributed and satellite-based decision-making algorithms. In this context, formal methods and tools are of paramount importance for designing and validating railway control systems, as witnessed by the large number of applications and case-studies carried out over three decades [14,6].

Alessandro Fantechi is one of the foremost scientists promoting the usage of formal methods in the railway domain. He considered the enhancement of the design and development of railway systems with formal methods along various ways: model-based testing and abstract interpretation [21,15], model checking [12,22], as well as quantitative analysis [4,2]. He is also an author of several surveys on the application of formal methods in the railway domain [6,13,1].

Besides his scientific contributions, Alessandro is also deeply involved in dissemination and steering activities. He is one of the originators of the DisCoRail workshop, dedicated to formal methods for distributed computing in future railway systems, which became in 2021 a track of the ISOLA conference series [11].

---

\* Institute of Engineering Univ. Grenoble Alpes

<sup>1</sup> [https://www.era.europa.eu/domains/infrastructure/european-rail-traffic-management-system-ertms\\_en](https://www.era.europa.eu/domains/infrastructure/european-rail-traffic-management-system-ertms_en)

He is regularly leading or participating to research projects aiming at the integration of formal methods in the railway domain, such as ASTRail [5], 4SECU-Rail [7], and ADVENTURE [3]. Alessandro is also active at the European level as a member of the scientific steering group of the Shift2Rail<sup>2</sup> initiative and of the Europe’s Rail Joint Undertaking<sup>3</sup>.

An important optimisation problem in railway transportation is train scheduling, which consists in producing a schedule for a set of  $n$  trains with fixed routes from origin to destination and going through particular sections of  $m$  tracks. By considering the trains as jobs, the tracks as machines, and the passings of trains through sections as tasks, the problem of minimising the travel time of all trains can be expressed as an instance of the well-known *job-shop scheduling problem*. Briefly, job-shop scheduling aims at producing a schedule to execute on  $m$  machines  $n$  jobs, each of which is a sequence of tasks  $t_i$ , each characterised by a pair  $(d_i, m_i)$  specifying the duration  $d_i$  of the task and the required machine  $m_i$ . The goal is to finish execution of all tasks as fast as possible, still respecting the order of tasks in each job and the fact that each machine can handle at most one task at a time. This combinatorial problem being NP-hard, an extensive amount of work has been devoted to devising various resolution approaches and heuristics for job-shop scheduling [9,31,25], which has many applications in manufacturing, equipment selection, chemicals, pharmaceuticals, healthcare, process planning, and—last, but not least—railway transportation.

Formulations of train scheduling as job-shop scheduling have been proposed for both the single-track case [28,24] and the double-track case [29], and extensive work has been dedicated to solve the corresponding job-shop instances [10,9]. In the context of train scheduling, one needs an additional constraint for the job-shop to represent the fact that a train passes atomically from one section to the next: this means that a free machine (track) must accept any available task (train passing) immediately. Stated otherwise, a machine must only be idle if there is no task requiring it. Alessandro’s contributions also encompass the optimisation of railway systems (including timetable improvements for delay minimisation), but using other approaches not related to the job-shop scheduling, such as stochastic models and simulation [20].

In this paper, we investigate the feasibility of tackling job-shop scheduling with the exhaustive state space exploration algorithms implemented in the CADP toolbox<sup>4</sup> [18]. Our work was carried out within the A-IQ Ready project<sup>5</sup>, which aims at devising intelligent autonomous electronic control systems based on crucial technologies, such as edge continuum orchestration for artificial intelligence, distributed collaborative intelligence, and quantum sensing. One of the use cases proposed in A-IQ Ready is related to indoor logistics, namely domestic robots navigating on tracks, for which optimal schedules must be produced.

<sup>2</sup> <https://rail-research.europa.eu/about-shift2rail/>

<sup>3</sup> <https://rail-research.europa.eu/about-europes-rail/europes-rail-structure-of-governance/scientific-steering-group/>

<sup>4</sup> <https://cadp.inria.fr>

<sup>5</sup> <https://www.aiqready.eu/>

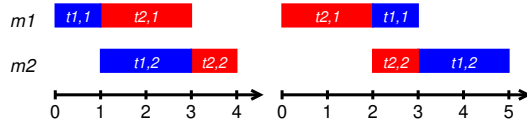


Fig. 1. Schedules for the Job-Shop Instance I2

Our state space algorithms being complete, they can provide guarantees for the optimality of a solution, an information sought by project partners to further assess the quality of their application of reinforcement learning to the search of solutions [30]. The idea was to use state space exploration to search for, and possibly prove the absence of, better solutions than those computed by reinforcement learning. We considered a freely available collection of job-shop scheduling instances with solutions<sup>6</sup>.

The rest of the paper is organised as follows. Section 2 introduces the common principles adopted for devising our formal models of job-shop scheduling. Sections 3, 4, 5, and 6 are devoted in turn to variants of the formal model, increasingly optimised for reducing the size of the state space. Section 7 describes how the models were exploited for state space generation and verification purposes. Section 8 discusses the heuristics adopted for reducing the state space. Finally, Section 9 gives some concluding remarks and directions for future work.

## 2 Common Modelling Principles

We experimented several encodings in the LNT language [19], a modern language combining the advantages of an imperative syntax with a formally defined semantics taken from process calculi. LNT is supported by the CADP toolbox, which provides tools to compile an LNT model into an LTS (Labelled Transition System). Common to all encodings is the idea to use a separate LNT process for each job and each machine, interacting via actions corresponding to the start and end of tasks. Using this encoding, different interleavings of these actions correspond to different schedules. Execution time of a schedule can be measured in two different ways: either using an offer indicating the current time or using a dedicated `TICK` action corresponding to a time-step. In any case, the LTS underlying the LNT model contains all possible schedules. Cutting the generation of the LTS at a given maximal time (or a maximal number  $k$  of `TICKS`) enables to search for schedules shorter than  $k$ .

To illustrate the various encodings in LNT, consider the following minimalistic instance I2, with two machines  $m_1$  and  $m_2$  and two jobs  $j_1 = [(1, m_1); (2, m_2)]$  and  $j_2 = [(2, m_1); (1, m_2)]$ . Figure 1 shows two possible schedules for this instance: the (optimal) schedule on the left requires four time steps, whereas the one on the right requires five time steps. Notice that both schedules assign a new task to a machine as soon as possible.

<sup>6</sup> <https://github.com/strassl/jssp-instances>

```

type machineID is
  M1, M2
end type

type jobID is
  J0, J1, J2
end type

type task is
  task (duration: nat, machine: machineID)
end type

type task_list is
  list of task
end type

```

**Fig. 2.** Data Type Definitions

A major modelling challenge is to avoid useless, clearly non-optimal schedules, so as not to bloat the LTS pointlessly. For instance, there is absolutely no point to leave all machines idling. A naive modelling, leaving machines and jobs free to interact, would include the case where no job starts executing.

Common to all our encodings is the representation of the available machines and jobs as enumerated data types, and tasks as records with two fields for the duration and the machine, as shown in Fig. 2; a machine executing the particular job J0 is considered to be free.

### 3 Single-Gate Model

In a first model, a single gate is used for all communications between jobs and machines. This has the advantage of passing all information in a single place, making it easy to enforce ordering constraints at each time step. However, it requires all LNT processes to accept rendezvous on the gate at every time step, and to share the current state of all jobs and machines with all processes. An additional gate `finish` is used to synchronise all LNT processes at the end when all tasks of all jobs are completed and the machines can stop their execution.

The behaviour of a machine is described by the LNT process shown in Fig. 3, and the behaviour of a job is described by the LNT process shown in Fig. 4. Both processes have a single visible gate `current`, specified between square brackets and with channel type `c` specifying that each rendezvous has seven offers: the current time, and for each machine a triple consisting of the currently executed job, the remaining time, and a Boolean indicating whether the machine has started the execution of this task in the current time step.

A machine has a variable parameter `m` corresponding to its identifier and local variables to hold the three values related to the three offers corresponding to

```

channel c is
  (tick: nat,
   current_job_1: jobID, time_remaining_1: nat, accepted_1: Bool,
   current_job_2: jobID, time_remaining_2: nat, accepted_2: Bool)
end channel

process machine [current: c] (m: machineID) is
  var current_job: jobID, time_remaining: nat, b: bool in
    current_job := J0;
    time_remaining := 0;
  loop
    if current_job != J0 then -- the machine is executing something
      case m in
        M1 => current (?any nat, current_job, time_remaining, false,
                     ?any jobID, ?any nat, ?any bool)
        | M2 => current (?any nat, ?any jobID, ?any nat, ?any bool,
                       current_job, time_remaining, false)
      end case;
      time_remaining := time_remaining - 1
    else -- the machine is free and ready to accept a new task
      case m in
        M1 => current (?any nat, ?current_job, ?time_remaining, ?b
                     ?any JobID, ?any nat, ?any bool)
          where (current_job == J0) =>
            ((time_remaining == 0) and not (b))
        | M2 => current (?any nat, ?any JobID, ?any nat, ?any bool,
                       ?current_job, ?time_remaining, ?b)
          where (current_job == J0) =>
            ((time_remaining == 0) and not (b))
      end case;
      if time_remaining > 0 then
        time_remaining := time_remaining - 1
      end if
    end if;
  if time_remaining == 0 then
    -- the execution of the current task is terminated and the machine
    -- becomes free
    current_job := J0
  end if
end loop
end var
end process

```

Fig. 3. Single-Gate Model: Machine

machine  $m$ .<sup>7</sup> The behaviour of a machine is essentially an infinite loop consisting of a conditional if statement depending whether the machine is free or not. If the machine is executing a task different from  $J_0$ , it constrains (in the rendezvous on gate `current`) the three offers (two to four for  $M_1$  and five to seven for  $M_2$ ) respectively (a) to the identifier of the job the tasks belongs to, (b) to the time remaining to execute the task, and (c) to the Boolean false indicating that the execution of the task did not start during this time step. Otherwise, the machine is free and ready to accept a new task for execution; if no task is assigned to it, then the remaining time must be 0 and the third offer must be false (enforced by the constraints on the rendezvous). Then, if necessary, the time remaining for the execution of the current task is decremented, and if job execution has finished, the machine is declared to be free by assigned it a task of job  $J_0$ .

The behaviour of a job is described similarly. Process `job` has two variable parameters: its identifier  $j$  and the initial list of tasks to be executed, each task being a pair consisting of a duration and the identifier of the machine, where the task has to be executed. The body of process `job` is essentially a sequence of two loops `L1` and `L2`. Loop `L1` describes the execution of the tasks of the job, executing the tasks of its task list parameter `tasks` one by one. Loop `L1` thus iterates over the list of tasks, selecting the first task of the list using pattern matching (case statement). For each task, it executes a while loop depending on the duration of the task, trying to start execution as soon as the requested machine becomes free. This is expressed by the nondeterministic choice (alt statement in the then branch of the if) with the two alternatives for a rendezvous on gate `current`: either the task starts its execution, or a task of another job is running: the job does not accept a rendezvous on `current` when the machine idles. A task is executed for as many steps as specified by its duration (which is decremented once per iteration of the while loop). When all tasks have been executed, loop `L2` permits other jobs to finish the execution of their tasks, enforcing only that no machine executes a task of job  $j$ . Using an alt statement permits to detect the situation where all machines are idle and to end the execution of the job.

The complete system is then described by the parallel composition of machines and jobs shown in Fig. 5. Both jobs and machines are grouped together in two separate parallel compositions. When all jobs have ended, their group executes a rendezvous on gate `finish`, which disrupts the execution of the machines (and the process `measure`, which simply enforces that the first offer of gate `current` increases by one each time, indicating elapsed execution time).

Overall, this LNT model has about 200 lines. The corresponding LTS is generated almost instantly and is shown in Fig. 6. Starting from the initial state (numbered 0 at the bottom), the two execution branches correspond to the two schedules shown in Fig. 1. The seven offers of the transitions labelled with a rendezvous on gate `current` indicate the progress of the execution. For instance, the transition “`CURRENT !3 !J0 !0 !FALSE !J1 !2 !TRUE`” between states 6 and 8

<sup>7</sup> It would also be possible to model each machine as a separate process—removing the need for the parameter  $m$  and the repeated `case`-constructs.

```

process job [current: c] (j: jobID, in var tasks: task_list) is
var k1, k2: jobID in
  loop L1 in
    case tasks var t: task in
      nil  $\Rightarrow$  break L1
    | cons (t, tasks)  $\Rightarrow$ 
      var time_remaining: nat, d: nat in
        time_remaining := t.duration;
        while time_remaining > 0 loop
          case t.machine of machineID in
            M1  $\Rightarrow$ 
              if time_remaining == t.duration then
                alt -- task t starts its execution
                  current (?any nat, j, time_remaining, true, ?k2,
                    ?any nat, ?any bool) where k2 != j;
                  time_remaining := time_remaining - 1
                [] -- a task of another job is running
                  current (?any nat, ?k1, ?d, ?any bool, ?k2,
                    ?any nat, ?any bool)
                    where (k1 != J0) and (d != 0) and
                      (k1 != j) and (k2 != j)
                end alt
              else -- task t is executing
                assert time_remaining > 0;
                current (?any nat, j, time_remaining, false, ?k2,
                  ?any nat, ?any bool) where k2 != j;
                time_remaining := time_remaining - 1
              end if
            | M2  $\Rightarrow$ 
              ... -- symmetric case
          end case
        end loop
      end var
    end case
  end loop;
loop L2 in
  alt
    -- no more executing jobs
    current (?any nat, J0, 0 of nat, false, J0, 0 of nat, false);
    break L2
  [] -- some jobs still executing
    current (?any nat, ?k1, ?any nat, ?any bool, ?k2, ?any nat,
      ?any bool)
      where (k1 != j) and (k2 != j) and ((k1 != J0) or (k2 != J0))
  end alt
end loop
end var end process

```

Fig. 4. Single-Gate Model: Job

```

process MAIN [current: c, finish: none] is
  par current, finish in
    par current in
      job [...] (J1, {task(1, M1), task(2, M2)})
      || job [...] (J2, {task(2, M1), task(1, M2)})
    end par;
    finish
  || disrupt
    par current in
      machine [...] (M1)
      || machine [...] (M2)
      || measure [...]
    end par
  by finish
end disrupt
end par
end process

```

Fig. 5. Single-Gate Model: Complete System

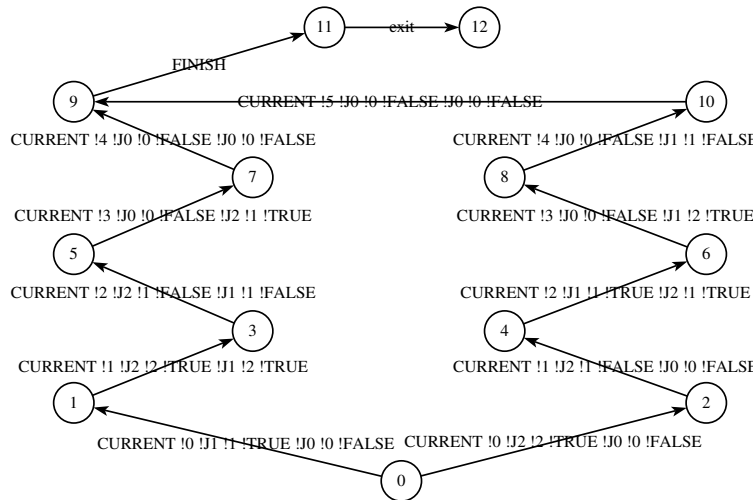


Fig. 6. LTS for the Single-Gate Model

indicates that in the third time step, machine  $m_1$  is idling, machine  $m_2$  started execution of a task from  $j_1$  that needs two time steps before completion.<sup>8</sup> The transition labelled `exit` between states 11 and 12 indicates successful termination, i.e., that the system finishes (without blocking).

A significant drawback of this modelling style is its lack of modularity: adding a machine or job requires to change all rendezvous in all LNT processes to take into account the additional offers (i.e., parameters describing exchanged data values), which also means changing all the constraints. Furthermore, the state space generation process in CADP uses a Petri net as intermediate format on which to apply optimisations. When constructing this Petri net, all possible rendezvous need to be considered. The many occurrences of gate `current` in the LNT processes `job` and `machine` induce a high number of possible combinations, leading to a large Petri net (and corresponding C code for the state space exploration). This is no difficulty for small instances, but larger instances become difficult to handle. Consider the instance I4 with four machines  $m_1, m_2, m_3$ , and  $m_4$  and four jobs (each with four tasks of duration one)<sup>9</sup>. Instance I4 has a single optimal solution requiring four time steps; the corresponding LTS has to be constructed compositionally and the corresponding C code requires almost 50 GB of RAM to be compiled by GCC.

## 4 Optimised Single-Gate Model

To circumvent some of the problems of the first single-gate model, namely to reduce the number of synchronisations and also avoid the computation of constraints that are provably not required for the next transition, we restructured the single-gate model of the previous section, leading to a second, optimised model. In this second model, a machine is described by the LNT process `machine` shown in Fig. 7, and a job by the LNT process `job` shown in Fig. 8. As before, both processes synchronise (with the same seven offers) on a single gate `current`.

The major difference is that process `machine` in Fig. 7 has a single occurrence of gate `current` (rather than three). For readability, the constraints on the rendezvous are grouped in a predicate (or Boolean function) `m_synchro`, and separated from the update of the local variables, grouped into a procedure (or function without result) `m_update`.

Similarly, the process `job` shown in Fig. 8 has only three occurrences of gate `current` (rather than eight), and groups the constraints into two predicates `job_synchro` (used when the job tries to submit a task for execution) and `j_execution` (used when a task of the job is executed). A third predicate `job_result` encapsulates the test whether job submission was successful. There is no need for a procedure to update the single local variable `d` counting the number of steps a task must still be executed.

<sup>8</sup> When generating the LTS, all characters are converted to upper case.

<sup>9</sup>  $j_1 = [(1, m_1); (1, m_2); (1, m_3); (1, m_4)]$ ,  $j_2 = [(1, m_2); (1, m_3); (1, m_4); (1, m_1)]$ ,  $j_3 = [(1, m_4); (1, m_1); (1, m_2); (1, m_3)]$ , and  $j_4 = [(1, m_3); (1, m_4); (1, m_1); (1, m_2)]$ .

```

process machine [current: c] (m: machineID) is
  var current_job: jobID, time_remaining: nat, k1, k2: jobID,
      d1, d2: nat, b1, b2: bool in
    current_job := J0;
    time_remaining := 0;
    loop
      current (?any nat, ?k1, ?d1, ?b1, ?k2, ?d2, ?b2)
        where m_synchro (m, current_job, k1, k2, d1, d2, b1, b2);
        eval m_update (m, k1, k2, d1, d2, b1, b2, current_job?,
                     time_remaining?);
        if time_remaining == 0 then
          -- nothing more to do for the current task (if any):
          -- the machine becomes free
          current_job := J0
        end if
      end loop
    end var
  end process

```

Fig. 7. Optimised Single-Gate Model: Machine

The overall parallel composition is the same as before (shown in Fig. 5). Taken altogether, this optimised LNT model has also about 200 lines of LNT and (almost instantly) yields the same LTS (shown in Fig. 6). However, the intermediate Petri net has fewer transitions.

For larger instances, the difference becomes significant. For the instance with four machines, four jobs with four tasks each, the optimised LNT model yields an intermediate Petri net with 137 transitions, which corresponds to a reduction of three orders of magnitude.

Still, the model is not modular: adding machines or jobs is cumbersome because it requires changes in almost all processes and predicates. Furthermore, because each transition synchronises all processes, all conditions need to be checked at each time step, the generation of the LTS is possible, but slow already for slightly larger instances as the one with four machines mentioned above (2.5 minutes for an LTS with 8 states and 7 transitions).

## 5 Two-Gate Model

To obtain a modular model, we let jobs and machines interact directly using two gates **assign** (to start execution of a task) and **done** (to indicate termination of the task). Similar to the single-gate models, an additional gate **finish** is used for the final synchronisation. The major advantage of this modelling style is that adding machines and/or jobs requires only changes in the overall parallel composition, because each rendezvous between a job and a machine carries only

```

process job [current: c] (j: jobID, in var tasks: task_list) is
var k1, k2: jobID, d, d1, d2: nat, b1, b2: bool in
  loop L1 in
    case tasks var t: task, rest: task_list in
      nil  $\Rightarrow$  break L1
    | cons (t, rest)  $\Rightarrow$ 
      current (?any nat, ?k1, ?d1, ?b1, ?k2, ?d2, ?b2)
        where job_synchro (j, t, k1, k2, d1, d2, b1, b2);
      if job_result (j, t.machine, k1, k2, b1, b2) then
        for d := t.duration - 1 while d > 0 by d := d - 1 loop
          current (?any nat, ?k1, ?d1, ?b1, ?k2, ?d2, ?b2)
            where j_execution (j, t.machine, d, k1, k2, d1, d2, b1, b2)
          end loop;
          tasks := rest
        end if
      end case
    end loop;
  loop L2 in
    -- allow other jobs to execute
    current (?any nat, ?k1, ?any nat, ?any bool, ?k2, ?any nat,
      ?any bool) where (k1 != j) and (k2 != j);
    if (k1 == J0) and (k2 == J0) then -- no more running jobs
      break L2
    end if
  end loop
end var end process

```

Fig. 8. Optimised Single-Gate Model: Job

```

process machine [assign, done: C, tick, finish: none] (m: machineID) is
var j: jobID, time: Nat in
  loop
    alt
      assign (?j, ?any Nat, ?time, m) where time <= max_job_duration;
      for null while time > 0 by time := time - 1 loop
        tick
      end loop;
      done (j, m)
    [] tick
    [] finish; stop
  end alt
end loop
end var end process

```

Fig. 9. Two-Gate Model: Machine

```

process job [assign, done: C, finish: none]
  (j: jobID, in var tasks: task_list) is
  loop
    case tasks var t: task, next_tasks: task_list in
      cons (t, next_tasks) =>
        assign (j, t.operation, t.duration, t.machine);
        done (j, t.machine);
        tasks := next_tasks
      | nil => finish; stop
    end case
  end loop
end process

```

Fig. 10. Two-Gate Model: Job

offers related to the two interacting processes. To measure execution time, all machines also synchronise on a gate tick modelling a discrete clock.

In this new model, a machine is described by the much simpler process shown in Fig. 9. It essentially executes a loop containing a nondeterministic choice between accepting a new task to be executed (rendezvous on gate `assign`), letting time progress (rendezvous on gate `tick`) and ending execution (rendezvous on gate `finish`). After accepting a task, the machine participates in as many rendezvous on gate `tick` as the duration of the task before a rendezvous on gate `done` indicating the corresponding job that the task has been executed.

A job is described by the process shown in Fig. 10. It iteratively executes the tasks from its list, each execution consisting in two rendezvous (on gate `assign` to start and on gate `done` to end the execution). When all tasks are executed, it synchronises will all other processes on gate `finish`.

```

process MAIN [assign, done: C, tick, finish: none] is
  par assign, done, finish in
    par
      job [...] (J1, job_1)
      || job [...] (J2, job_2)
    end par
  || par tick, finish in
      machine [...] (M1)
      || machine [...] (M2)
    end par
  end par
end process

```

**Fig. 11.** Two-Gate Model: Complete System

The overall parallel composition is shown in Fig. 11. Note that only the machines synchronise on gate `tick` and that there is no need to use the `disrupt` statement. Also, adding another job or machine is straightforward.

With about 100 lines of LNT, this model is significantly shorter than both single-gate models, but generates (almost instantly) an LTS with 55 states and 84 transitions (when minimised for strong bisimulation), which is significantly larger than the one shown in Fig. 5. Hiding the rendezvous on gate `done` and minimising for branching bisimulation yields an LTS with 28 states and 51 transitions, difficult to reduce further, because the rendezvous on gate `assign` are necessary to extract the scheduling of the task executions, and the rendezvous on gate `tick` are necessary to detect simultaneously started tasks. Besides a shorter source code, this modelling style is better suited to the CADP tools, because it avoids the computation of complex constraints in the data type and favours the selection of choices in the interleavings of the processes. Hence, state space generation is much more efficient: For the instance with four machines and four jobs with four tasks each, the complete LTS (16,550 states and 36,890 transitions) can be generated in about a second.

Actually, this two-gate model was our first modelling attempt, but was abandoned because it violated the constraint that a (free) machine must execute an available task immediately. Unfortunately, it is impossible to express this constraint in LNT, because LNT has no means to express priorities on alternatives in a choice.<sup>10</sup> Furthermore, in the two-gate model, each state of the LTS corresponding to a state with at least one free machine has a `TICK` self-loop, introducing a choice between executing a new task immediately or delaying the execution for one `TICK`. When considering larger instances with more machines, jobs, and tasks, these choices induce a huge number of states and transitions,

<sup>10</sup> Notice that a monolithic model, encoding everything using data-types, functions and a single process determining for each time instant the tasks executed by all machines, might have been a solution to express such priorities.

growing exponentially. The single-gate models were our first try to keep the LTS size under control.

As a second attempt to cope with larger models, we experimented with the compositional state space generation tools available in CADP. The idea is to generate each process separately, minimise it and then compute the overall composition, applying priorities, cutting superfluous execution branches. Due to the high number of possible triples (job, task identifier, duration) to be taken into account every time the machine is free, the naive compositional approach faces the unfortunate situation where a single process is similar in size to the complete state space. However, for each job, the corresponding LTS can be easily generated: thus the exact set of possible assignments emitted by the job is known. Using this set as interface (by means of a chaos automaton with a single state and a self-looping transition for each label in the set), semi-composition [23] enables the generation of the LTSs for the machines. When composing the LTSs for jobs and machines, the application of priority rules further reduces the size of the overall LTS. A first rule enforces that all actions related to job execution have a higher priority than the progression of time; between two `TICK` transitions, it is sufficient to consider a single ordering of transitions: hence we used priority rules to ensure that `DONE` is handled before `ASSIGN` (to have all machines available when choosing the next jobs to execute) and that `DONE` and `ASSIGN` are ordered according to increasing machine numbers. We automated these generation, reduction, and composition steps using an SVL [17] script, which takes advantage of the advanced smart reduction heuristics [8] (implemented in the SVL tool) to devise the order of the various steps.

For small instances, there is no gain in execution time, due to the largely increased number of calls to the various CADP tools: for the instance I2 with two machines, two jobs with two tasks each, the generation takes a total of 90 seconds. However, the resulting LTS is much smaller: 26 states and 26 transitions, which further reduces—after hiding `DONE` transitions and minimising for branching bisimulation—to 17 states and 17 transitions. Although this is as small as it can get, it is clearly larger than the LTS shown in Fig. 5, because each `TICK` and each assignment of a task to a machine yields a separate transition rather than merging all assignments of a same time step in a single transition (as in Fig. 5).

For the instance I4 with four machines, direct generation takes less than two seconds, and yields an LTS with 16050 states and 36754 transitions (minimised for strong bisimulation), whereas compositional generation takes 27 seconds and yields an LTS with 22 states and 21 transitions. Notice that both LTSs correspond to the single optimal solution of I4: the difference is due to the fact that direct generation considers all interleavings, whereas the priority rules in the compositional approach leave only a single interleaving.

## 6 Compact Two-Gate Model

To further reduce the number of transitions in the final LTS, we experimented with a compact version of the two-gate model, where sequences of `TICK`-

```

process machine [assign, done: C, tick: T, finish: none]
    (m: machineID) is
var j: jobID, time, t: Nat in
    loop
    alt
    assign (?j, ?any Nat, ?time, m) where time <= max_job_duration;
    while time > 0 loop
    tick (?t) where (t > 0) and (t <= time);
    time := time - t
    end loop;
    done (j, m)
    [] tick (?t) where (t > 0) and (t <= max_job_duration)
    [] finish; stop
    end alt
    end loop
end var end process

```

**Fig. 12.** Compact Two-Gate Model: Machine

transitions are combined into a single TICK-transition with an offer corresponding to the length of the sequence. This approach is interesting for instances with long task durations (the durations in the benchmark require most frequently two digits). The corresponding LNT code for a machine is shown in Fig. 12.

The condition “time <=max\_job\_duration” in the rendezvous on gate assign is not necessary when generating the LTS for the complete system, because a single value is provided by the corresponding job; however, the condition helps reducing the size of the LTS of a machine taken separately when computing the overall LTS compositionally.

Direct generation yields (in about 3 seconds) an LTS with 53 states and 970 transitions (25 states and 934 transitions after hiding DONE and minimising for branching bisimulation). The surprising increase in the number of transitions is due to the  $n!$  choices to break up a sequence of  $n$  TICK transitions.

When using the compositional generation, an additional priority rule giving priority to the largest time steps avoids this increase. This priority rule specifying the order of all considered values for TICK transitions is generated in the SVL script by ordering all labels of the form “TICK !n” present in all machines. Overall, compositional generation is also improved: As expected, the LTS is smaller, having 23 states and 23 transitions (15 states and 15 transitions after hiding DONE and minimising for branching bisimulation), because the two sequences of two TICK transitions are merged into a single TICK transition. Also, the execution time is divided by more than four (it is reduced to 17 seconds).

## 7 Exploitation of the Model(s)

The models presented in the previous sections can be used to study the job-shop scheduling problem using various verification techniques. Generating the complete LTS is interesting to study all possible solutions of a given instance. If the goal is to search for a single solution, it is not necessary to construct the complete LTS, but rather search the solution in the LTS using on-the-fly techniques, e.g., by model checking the following MCL [27] formula:

`< true*. FINISH > true`

This formula is necessarily satisfied, because there is always the trivial schedule executing all jobs in sequence. Thus, the diagnostic (here, a witness transition sequence) generated by the model checker describes a possible solution. Another applicable on-the-fly tool is TESTOR [26], which generates conformance test(s) for a test purpose requesting to observe a transition `FINISH`; depending on the options, this allows to compute a single solution (a test case) or all solutions (the so-called complete test graph).

When the goal is to improve a known solution, it is sufficient to bound the LTS exploration to a given depth, either by modifying the LNT model (e.g., by adding a process limiting the number of `TICKS`), by using exploration tools limited to a given depth, or by bounding the number of `TICKS` in the property or test purpose. This on-the-fly exploration can be applied on the complete model, as well as on the compositionally constructed model: the latter has the advantage of enabling the various priority rules to be taken into account.

The generation of small instances of the job-shop benchmarks was successful, with reasonable performance. For example, the LTS for the instance Fisher-Thompson FT06 has 68816 states and 72543 transitions. Unfortunately, larger instances (more than six machines, jobs, and tasks) of the benchmarks could not be handled. Considering the instance Fisher-Thompson FT10 with ten machines and ten jobs with ten tasks each, experiments with compositional state space generation considered adding jobs one by one (because each job assigns tasks to all ten machines, the machines must always be included). Table 1 gives the LTS sizes for four to eight jobs (of the ten jobs of FT10); all LTSs but the one for eight jobs have been minimised for strong bisimulation. Generation of the larger LTSs took advantage of distributed generation tools, using up to sixteen cores. Generation of the LTS for nine jobs was interrupted due to lack of memory (it required more 900 GB of RAM).

**Table 1.** State-Space Generation Statistics for Instance FT10

#jobs	states	transitions
4	1,260	1,281
5	9,234	9,470
6	153,358	159,542
7	4,059,402	4,250,087
8	733,172,821	772,979,502

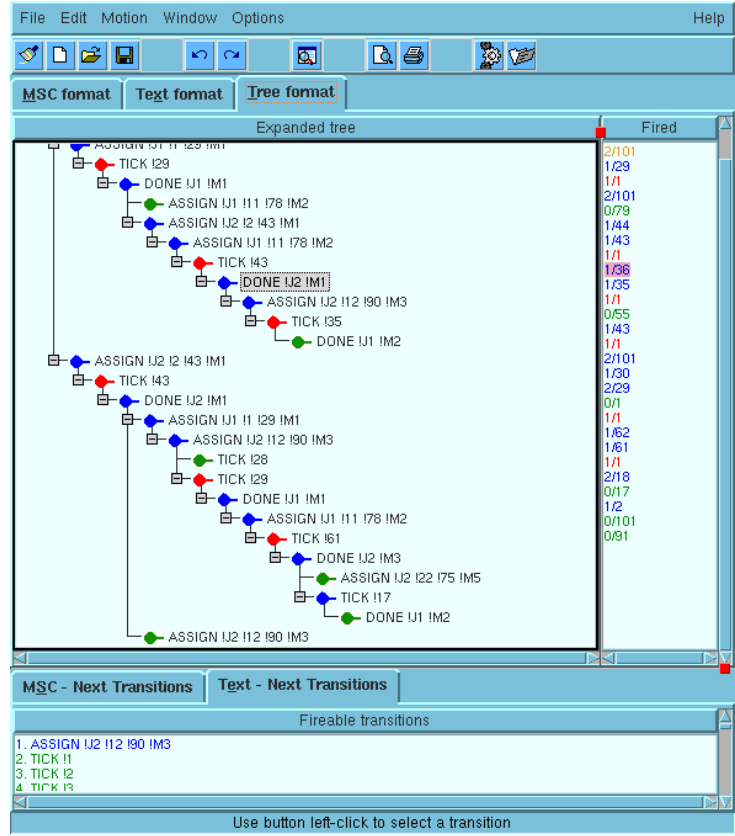


Fig. 13. Interactive Simulation of Instance FT10

Even if generating the complete LTS is out of reach due to the combinatorial explosion inherent to the job-shop scheduling problem, the model is still usable: Fig. 13 is a screenshot of the step-by-step simulator for the instance FT10, running smoothly on a standard laptop.

## 8 Correction of the Heuristics

The models presented so far were all based on an heuristics, required for the application to railway scheduling, to execute tasks as soon as possible, i.e., if a machine  $m$  is free and there is some job searching to execute a task  $t$  on  $m$ , then  $t$  is immediately assigned to  $m$ . This heuristics greatly cuts the size of the LTS and was a major reason motivating the switch from the initial two-gate model to the more complex single-gate model. Unfortunately, this heuristics is incorrect for general job-shop scheduling problems, as illustrated by the following instance I3 with three machines  $m_1$ ,  $m_2$ , and  $m_3$ , and

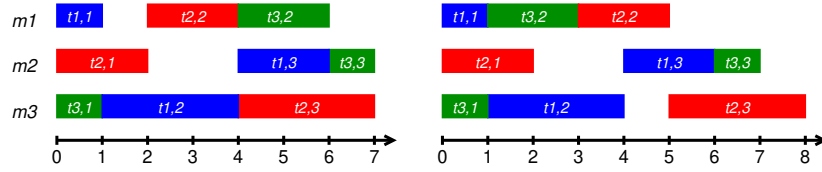


Fig. 14. Schedules for the Job-Shop Instance I3

three jobs  $j_1 = [(1, m_1); (3, m_3); (2, m_2)]$ ,  $j_2 = [(2, m_2); (2, m_1); (3, m_3)]$ , and  $j_3 = [(1, m_3); (2, m_1); (1, m_2)]$ .

In the optimal schedule (on the left of Fig. 14) at time instant one, machine  $m_1$  remains free, although the second task of job  $j_3$  could be assigned. However, the previous heuristics yields the non-optimal schedule on the right of Fig. 14. The problem is that job  $j_2$  constitutes the critical path, and must not be delayed.

Changing the heuristics for the two-gate models is easy: direct generation produces all possible interleavings, and for compositional generation it is sufficient to adapt the priority rules such that `assign` has a smaller priority than `tick`, enabling machines to idle even in presence of tasks ready to be assigned. Note that although this adds `tick` self-loops to almost all states, the other priority rules still greatly reduce the size of the state space. Concretely, direct generation of instance I3 yields an LTS with 1219 states and 2082 transitions, whereas compositional generation using the corrected priority rules yields an LTS with 38 states and 51 transitions (26 states and 39 transitions after hiding `done` and minimising for branching bisimulation), containing the optimal schedule.

Correcting the single-gate models is more involved, because it requires one to devise a constraint checking whether an assignment would delay the critical path. Taking into account that the single-gate models suffer already from the heavy computation of the existing constraints, adding such a constraint was not further investigated.

## 9 Conclusion

We studied how the well-known job-shop scheduling problem can be formulated and solved using modern formal description languages and their associated verification tools. We devised and progressively refined several models of the job-shop problem in LNT, a formal language with user-friendly syntax and process algebraic semantics. The final two-gate model, in which jobs and machines interact via the gates `assign` and `done` (indicating the start and termination of a task, respectively) is modular and compact, allowing one to easily set up new configurations of jobs and machines. The corresponding LTS was made as small as possible by using heuristics (implemented using the priority rules of the SVL language), still being guaranteed to contain the optimal schedules, which can be found using the on-the-fly model checker of the CADP toolbox.

Due to the inherent combinatorial complexity of job-shop scheduling, increasing the number of machines and the size of jobs leads to state-space explosion. However, our two-gate LNT model can be used as basis to implement and experiment various heuristics to guide the search towards optimal schedules, by compositionally adding constraints in the LNT model and exploring the LTS on the fly using the OPEN/CÆSAR environment [16].

**Acknowledgments.** Part of this work has been supported by the Schlumberger Foundation and by the A-IQ Ready project, which receives funding within the Chips JU (the Public-Private Partnership for research, development and innovation under Horizon Europe) and National Authorities under grant agreement no. 101096658.

## References

1. Basile, D., ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F., Piattino, A., Trentini, D., Ferrari, A.: On the industrial uptake of formal methods in the railway domain - A survey with stakeholders. In: Proceedings of the 14th International Conference on Integrated Formal Methods (IFM'2018), Maynooth, Ireland. Lecture Notes in Computer Science, vol. 11023, pp. 20–29. Springer (September 2018). [https://doi.org/10.1007/978-3-319-98938-9\\_2](https://doi.org/10.1007/978-3-319-98938-9_2)
2. Basile, D., ter Beek, M.H., Giandomenico, F.D., Fantechi, A., Gnesi, S., Spagnolo, G.O.: 30 years of simulation-based quantitative analysis tools: A comparison experiment between möbius and uppaal SMC. In: Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles (ISoLA'2020), Rhodes, Greece. Lecture Notes in Computer Science, vol. 12476, pp. 368–384. Springer (October 2020). [https://doi.org/10.1007/978-3-030-61362-4\\_21](https://doi.org/10.1007/978-3-030-61362-4_21)
3. Basile, D., Beek, M.H.T., Carnevali, L., Chiaradonna, S., Giandomenico, F.D., Fantechi, A., Gori, G.: An integrated perspective on the evaluation of complex railway systems. In: Proceedings of the 12th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Application Areas (ISOLA'2024). Lecture Notes in Computer Science, vol. 15223, pp. 190–207. Springer (2024). [https://doi.org/10.1007/978-3-031-75390-9\\_13](https://doi.org/10.1007/978-3-031-75390-9_13)
4. Basile, D., Fantechi, A., Rosadi, I.: Formal analysis of the UNISIG safety application intermediate sub-layer - applying formal methods to railway standard interfaces. In: Proceedings of the 26th International Conference on Formal Methods for Industrial Critical Systems (FMICS'2021), Paris, France. Lecture Notes in Computer Science, vol. 12863, pp. 174–190. Springer (August 2021). [https://doi.org/10.1007/978-3-030-85248-1\\_11](https://doi.org/10.1007/978-3-030-85248-1_11)
5. ter Beek, M.H., Borälv, A., Fantechi, A., Ferrari, A., Gnesi, S., Löfving, C., Mazzanti, F.: Adopting formal methods in an industrial setting: The railways case. In: Formal Methods – The Next 30 Years. Lecture Notes in Computer Science, vol. 15240, pp. 762–772. Springer (2019). [https://doi.org/10.1007/978-3-030-30942-8\\_46](https://doi.org/10.1007/978-3-030-30942-8_46)
6. ter Beek, M.H., Fantechi, A., Gnesi, S.: Formal methods for industrial critical systems - 30 years of railway applications. In: The Combined Power of Research,

- Education, and Dissemination - Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday. *Lecture Notes in Computer Science*, vol. 15240, pp. 327–344. Springer (2025). [https://doi.org/10.1007/978-3-031-73887-6\\_21](https://doi.org/10.1007/978-3-031-73887-6_21)
7. Belli, D., Fantechi, A., Gnesi, S., Masullo, L., Mazzanti, F., Quadrini, L., Trentini, D., Vaghi, C.: The 4securail case study on rigorous standard interface specifications. In: *Proceedings of the 28th International Conference on Formal Methods for Industrial Critical Systems (FMICS'2023)*. *Lecture Notes in Computer Science*, vol. 14290, pp. 22–39. Springer (2023). [https://doi.org/10.1007/978-3-031-43681-9\\_2](https://doi.org/10.1007/978-3-031-43681-9_2)
  8. Crouzen, P., Lang, F.: Smart Reduction. In: Giannakopoulou, D., Orejas, F. (eds.) *Proceedings of Fundamental Approaches to Software Engineering (FASE'11)*, Saarbrücken, Germany. *Lecture Notes in Computer Science*, vol. 6603, pp. 111–126. Springer (Mar 2011)
  9. Dauzère-Pérès, S., Ding, J., Shen, L., Tamssaouet, K.: The flexible job shop scheduling problem: A review. *European Journal of Operational Research* **314**(2), 409–432 (April 2024). <https://doi.org/10.1016/j.ejor.2023.05.017>
  10. Fang, W., Yang, S., Yao, X.: A survey on problem models and solution approaches to rescheduling in railway networks. *IEEE Transactions on Intelligent Transportation Systems* **16**(6), 2997–3016 (2015). <https://doi.org/10.1109/TITS.2015.2446985>
  11. Fantechi, A., Gnesi, S., Haxthausen, A.: Formal methods for distributed computing in future railway systems. In: *Proceedings of the 12th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Application Areas (ISoLA'2024)*, Crete, Greece. *Lecture Notes in Computer Science*, vol. 15223, pp. 109–111. Springer (October 2024). [https://doi.org/10.1007/978-3-031-75390-9\\_7](https://doi.org/10.1007/978-3-031-75390-9_7)
  12. Fantechi, A., Haxthausen, A.E., Nielsen, M.B.R.: Model checking geographically distributed interlocking systems using UMC. In: *Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'2017)*, St. Petersburg, Russia. pp. 278–286. IEEE Computer Society (March 2017). <https://doi.org/10.1109/PDP.2017.66>
  13. Ferrari, A., ter Beek, M.H., Mazzanti, F., Basile, D., Fantechi, A., Gnesi, S., Piattino, A., Trentini, D.: Survey on formal methods and tools in railways: The astrail approach. In: *Proceedings of the 3rd International Conference on Reliability, Safety, and Security of Railway Systems (RSSRail'2019)*, Lille, France. *Lecture Notes in Computer Science*, vol. 11495, pp. 226–241. Springer (June 2019). [https://doi.org/10.1007/978-3-030-18744-6\\_15](https://doi.org/10.1007/978-3-030-18744-6_15)
  14. Ferrari, A., Beek, M.H.T.: Formal methods in railways: A systematic mapping study. *ACM Computing Surveys* **55**(4) (Nov 2022). <https://doi.org/10.1145/3520480>
  15. Ferrari, A., Magnani, G., Grasso, D., Fantechi, A., Tempestini, M.: Adoption of model-based testing and abstract interpretation by a railway signalling manufacturer. *International Journal on Embedded and Real-Time Communication Systems* **2**(2), 42–61 (2011). <https://doi.org/10.4018/JERTCS.2011040103>
  16. Garavel, H.: OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In: Steffen, B. (ed.) *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, Lisbon, Portugal. *Lecture Notes in Computer Science*, vol. 1384, pp. 68–84. Springer, Berlin (Mar 1998), full version available as INRIA Research Report RR-3352

17. Garavel, H., Lang, F.: SVL: a Scripting Language for Compositional Verification. In: Kim, M., Chin, B., Kang, S., Lee, D. (eds.) Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), Cheju Island, Korea. pp. 377–392. Kluwer Academic Publishers (Aug 2001), full version available as INRIA Research Report RR-4223
18. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. Springer International Journal on Software Tools for Technology Transfer (STTT) **15**(2), 89–107 (Apr 2013)
19. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: Katoen, J.P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10500, pp. 3–26. Springer (Oct 2017)
20. Giandomenico, F.D., Fantechi, A., Gnesi, S., Itria, M.L.: Stochastic model-based analysis of railway operation to support traffic planning. In: Proceedings of the 5th International Workshop on Software Engineering for Resilient Systems (SERENE'2013), Kiev, Ukraine. Lecture Notes in Computer Science, vol. 8166, pp. 184–198. Springer (October 2013). [https://doi.org/10.1007/978-3-642-40894-6\\_15](https://doi.org/10.1007/978-3-642-40894-6_15)
21. Grasso, D., Fantechi, A., Ferrari, A., Becheri, C., Bacherini, S.: Model based testing and abstract interpretation in the railway signaling context. In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'2010), Paris, France. pp. 103–106. IEEE Computer Society (April 2010). <https://doi.org/10.1109/ICST.2010.44>
22. Haxthausen, A.E., Fantechi, A.: Compositional verification of railway interlocking systems. Formal Aspects of Computing **35**(1), 1–46 (2023). <https://doi.org/10.1145/3549736>
23. Krimm, J.P., Mounier, L.: Compositional State Space Generation from LOTOS Programs. In: Brinksma, E. (ed.) Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), University of Twente, Enschede, The Netherlands. Lecture Notes in Computer Science, vol. 1217, pp. 239–258. Springer (Apr 1997), extended version with proofs available as Research Report VERIMAG RR97-01
24. Lange, J., Werner, F.: Approaches to modeling train scheduling problems as job-shop problems with blocking constraints. Journal of Scheduling **21**(2), 191–207 (2018). <https://doi.org/10.1007/S10951-017-0526-0>
25. Li, X., Guo, X., Tang, H., Wu, R., Wang, L., Pang, S., Liu, Z., Xu, W., Li, X.: Survey of integrated flexible job shop scheduling problems. Computers & Industrial Engineering **174**(108786) (2022). <https://doi.org/10.1016/j.cie.2022.108786>
26. Marsso, L., Mateescu, R., Serwe, W.: TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation. In: Beyer, D., Huisman, M. (eds.) Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), Thessaloniki, Greece. Lecture Notes in Computer Science, vol. 10806, pp. 211–228. Springer (Apr 2018)
27. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) Proceedings of the 15th International Symposium on Formal Methods (FM'08), Turku, Finland. Lecture Notes in Computer Science, vol. 5014, pp. 148–164. Springer (May 2008)
28. Oliveira, E., Smith, B.M.: A combined constraint-based search method for single-track railway scheduling problem. In: Proceedings of the 10th Portuguese Conference on Artificial Intelligence (EPIA'2001), Porto, Portugal. Lecture Notes in

- Computer Science, vol. 2258, pp. 371–378. Springer (December 2001). [https://doi.org/10.1007/3-540-45329-6\\_36](https://doi.org/10.1007/3-540-45329-6_36)
29. Palgunadi, S., Supraba, D., Harjito, B.: Job-shop scheduling model for optimization of the double track railway scheduling: (case study: Solo-yogyakarta railway network). In: Proceedings of the International Conference on Information & Communication Technology and Systems (ICTS'2016), Surabaya, Indonesia. pp. 90–95. IEEE Computer Society (2016). <https://doi.org/10.1109/ICTS.2016.7910279>
  30. Vivekanandan, D., Wirth, S., Karlbauer, P., Klarmann, N.: A reinforcement learning approach for scheduling problems with improved generalization through order swapping. *Machine Learning & Knowledge Extraction* **5**, 418–430 (April 2023). <https://doi.org/10.1145/3520480>
  31. Xiong, H., Shi, S., Ren, D., Hu, J.: A survey of job shop scheduling problem: The types and models. *Computers & Operations Research* **142**(105731) (2022). <https://doi.org/10.1016/j.cor.2022.105731>