Distributed Constraint-based Railway Simulation

Hans Schlenker

Fraunhofer FIRST, Kekuléstr. 7, 12489 Berlin, Germany
hans.schlenker@first.fraunhofer.de

Abstract. In railway simulation, given timetables have to be checked against various criteria, mainly correctness and robustness. Most existing approaches use classical centralized simulation techniques. This work goes beyond that in two main aspects: We use constraint satisfaction to get rid of dead lock problems and the simulation is done distributed for better performance. This should make it possible to solve very large railway simulation problems.

1 Introduction

1.1 Railway Simulation

A railway system [11, 15] consists of a set of stations, a network of tracks that connects the stations, a set of trains, and a timetable. The timetable assigns each train and each station this train must pass two points in time: when the train is scheduled to reach the station (*arrival*) and when to leave (*departure*). Each train moves from one station to the next along the network built by the tracks. Signals and additional devices like train-end-detectors ensure safety on the tracks. Blocks are subnets, delimited by signals and train-end-detectors. The railway's today's fundamental safety rule, which applies to all current long-distance railways, is: *There may never be more than one train within one block.*

The issue of *railway simulation* is to virtually let trains run through the network and to check whether the timetable is satisfiable (*correctness*) or stable against perturbations (*robustness*), always under the given safety restrictions. Note that the timetable to be checked is given in advance. *Building* a timetable is a different problem, though sharing parts with our case.

There exist some fundamental approaches to simulate physical systems [4, 7]: continuous vs. discrete event simulation and time driven vs. event driven simulation. Common to all approaches is that the real world system is described in terms of states and events. In continuous event simulation, "state changes occur continuously in time, while in a discrete simulation, the occurrence of an event is instantaneous and fixed to a selected point in time"[7]. Also according to [7], any continuous model can be converted to an equivalent discrete model, such that discrete event simulation can be used to model every physical system. It is obvious that discrete event simulation can be done naturally on modern (discrete) computer architectures.

In time driven simulation, the simulator looks at the virtual system in discrete points in time. You can think of this simulation following a given clock pulse. Event driven simulation, on the other hand, uses an event list, that stores all future events. When an event is processed (the one in the list with the lowest time stamp), this may generate new events which are inserted into the list according to their future time stamp (e.g. [4]). Event driven simulation has already been successfully applied to traffic simulation problems (e.g. [14]).

Distributed event simulation is an extension to event driven simulation. Here, the system to be simulated is divided into parts, which are evaluated on physically distinct (computer) nodes. Events are sent through the simulator's network from where they occur to where they make an impact on. Milestone works on discrete event simulation (e.g. [4, 8]) deal with methods how to asynchronously let some nodes run into the (simulated) future while others still treat the (simulated) past.

1.2 Constraint-Based Railway Simulation

In constraint-based simulation (CBS), we also use discrete time events, but use a completely different modeling: The system to be simulated is described as one complex constraint satisfaction problem (CSP) [10]. This CSP is then solved using well-known and newly adapted propagation and search techniques. A solution to the CSP is finally mapped into a description of a simulation run.

CBS basically works as follows. The railway network is mapped into an abstract discrete model: It is divided into *blocks*, while each *real track section* may belong to more than one block. A block is then the atomic exclusion unit: In no event, one block may be occupied by more than one train at the same time. The way of a train through the network is divided into blocks such that the concatenation of all parts makes up the whole way of the train from its source to its destination. Figure 1 depicts this approach. The blocks are modeled using constraint variables for the starting times and durations, which are connected through arithmetic equations. The timetable is given by minimal departure times for each train and each block (possibly being 0). Note that the arrival times are not directly given as constraints! The fundamental safety law is ensured using well-known resource constraints (like *cumulative* or *diffn* [1]). Assigning start and duration times to each part with respect to its block gives then rather directly a solution to the simulation problem.

The big advantage of this approach is that dead lock situations are detected very early: constraint propagation does this for us. We can thus prevent most situations where a number of trains jointly lock up parts of the railway network. This is the case for example when one track serves both directions (which is a common situation in Germany) and two opposed trains stand head by head (in theory, a train can't go backwards). This special one-track case has been exhausted in [5].

In contrast to classical approaches, ours does not have a continuous advancing simulation time: Propagation may run from the future to the past. This is untypical compared to classical simulation. Fortunately, this approach has

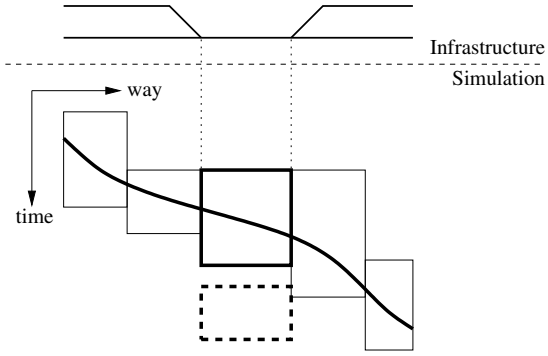

Fig. 1. Constraint-Based simulation: The way of the train through time and space (thick line) is surrounded by blocks (rectangles) that are related to the infrastructure (e.g. the thick bounded one and the single track section above). The fundamental safety law requires that corresponding rectangles never overlap: The dashed bounded block of another train must not overlap the thick bounded block of this train on the same track section.

an analogon in our real-world problem: The actual movements of all trains are guided by a real-time train management. This management also looks into the future, when determining the actual train's scheduling, for example to decide which train may leave a certain station immediately, and which one has to wait. Constraint based simulation equivalently uses this information to detect and avoid dead-lock cases. We call this approach therefore *look-ahead* simulation.

1.3 Distributed Constraint-Based Railway Simulation

In general, there are several reasons for *distributed* problem solving: reliability, privacy and social boundaries, and performance and load balancing (see e.g. [9]). In the research and development project SIMONE [13], this work is part of, we want to solve very large simulation problems. Therefore, *performance* (or *scalability*) is our main motivation for distributing the simulation.

In distributed simulation, the simulation problem has to be divided into subparts, which are then simulated in several computing nodes. A meta-algorithm conducts the cooperative solving process. There are several general purpose concepts to cooperatively solve distributed constraint problems: e.g. [3, 16, 19]. Most of them are characterized by distributed search or propagation. I, however, favor a more application-oriented approach: distribution is done on the application level rather than on the constraint network level. This avoids vast communication overhead due to micro-propagation between different computers over some network. Propagation in its very constraint programming sense (e.g. looking-ahead, forward-checking [10]) is done only within one simulator node. Coordination is done by the meta-algorithm.

Regarding railway simulation, there is currently one important work dealing with *distributed* simulation: [12]. Here, a number of local discrete event simulations jointly compute a global simulation. Each local node is responsible for a part of the network and the nodes exchange train information for trains leaving a subnet and entering another subnet. The simulation clocks are synchronized such that all computing nodes know about the global simulation time. This is also the main drawback of this approach: There is usually a lot of synchronization information to be communicated. This bottleneck greatly obstructs scalability of the algorithm.

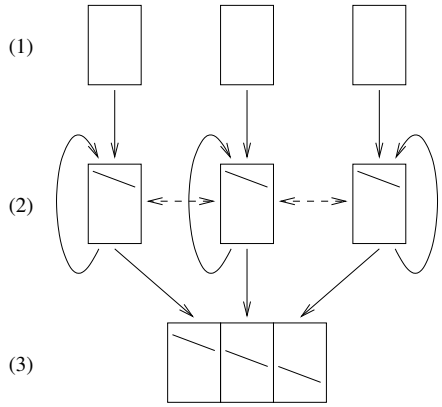

Fig. 2. The DRS algorithm.

2 Algorithm

Figure 2 depicts my distributed railway simulation algorithm DRS. Initially (1), the global simulation problem is divided into parts, which are distributed on several computing nodes. There, solutions for the local problems are computed (2). Information regarding the sub-problems borders is then communicated between sub-problems' neighbors (in Figure 2: the dashed arrows). While the local problems are globally inconsistent – what can be detected locally – the local simulations are iterated, taking into account the neighbors information. Finally (3), a globally consistent solution is achieved.

2.1 Problem Distribution

Each railway simulation problem is kind of naturally distributed: Its track network is spread spatially. Therefore, the natural way to decompose and distribute the simulation problem is along the railway network: We cut the network into subnets.

In our real-world problem, the data is already pre-partitioned. Each track section belongs to exactly one so-called *operating site* (OS, German *Betriebsstelle*). So, each OS o consists of a number s_o of track sections, and to each of o's neighbors o' a number $t_{o,o'}$ of crossing trains. The OSs together with their neighboring relation form a network. The nodes in this network are weighted by s_o, and the edges between two nodes o and o' are weighted by $t_{o,o'}$. We also assume some given k that relates to the number of available computing nodes.

This network is then cut into k parts such that the sum of s_o in all parts is uniformly distributed (there is no part with a sum that is by far greater than the other sums), and the edge cut is minimized. The edge cut is the sum of the weights of cut edges, i.e. edges between nodes that do not belong to the same part. This partitioning is the basis for the problem distribution. The described optimizations cause that the size of the different sub-problems is uniformly distributed and therefore the workload for the computing nodes is balanced, and that the number of crossing trains that have to be communicated between different nodes is minimized, reducing communication and – as we will see – recomputation costs.

2.2 Iterations

The local sub-problems are given by sub-nets of the global track network. In each local iteration, the sub-problem is simulated using the above described constraint-based simulation technique. As soon as such a simulation is finished, the entering and leaving times of all crossing trains are communicated to the node's neighbors. Thus, each node can locally determine which train's simulation is globally inconsistent.

While there are inconsistent trains, the local simulation CSP is extended, such that the local times for incoming and outgoing trains must be *greater or equal* the times given by the neighbors. Then, a new solution to the updated CSP is computed, leading to new entering and leaving times. All *new times that are different* from previous solutions, are considered (globally) inconsistent and are therefore communicated to the neighbors. Since each train that crosses parts may trigger re-computations of local simulations, it is obvious that their number should be kept minimal.

Note that the computations of the local simulations can be done interleaved: While one node a is computing, its neighbor b can be finished, sending a the new crossing data. Some moments before, b's other neighbor c finished its work and sent some connection data to b so that b now immediately could recompute its local simulation, taking into account c's work. He would not wait until a finished. This approach makes maximal use of the available computing resources. It is, however, not deterministic: Two (global) simulation runs on the same problem may not produce the same results, depending on the order in which the local simulation jobs have been executed.

This issue can be solved by synchronizing the simulation processes: All local simulations wait for each other after they have finished their local computations. And when all are finished, all of them communicate and then all of them

recompute (in case it is necessary). This algorithm is deterministic and therefore always yields the same results. The drawback here is that the computational resources are used less optimally.

2.3 Theory

Many distributed problem solving algorithms do not terminate on their own in all cases. Mostly, there has to be some algorithm-external termination detection. DRS, however, terminates always, i.e. it always finds a solution (if one exists) in finite time. This – together with correctness – can be proven theoretically. The proof, however, does not fit here (nor on this page's margins). The curious reader should consult [17].

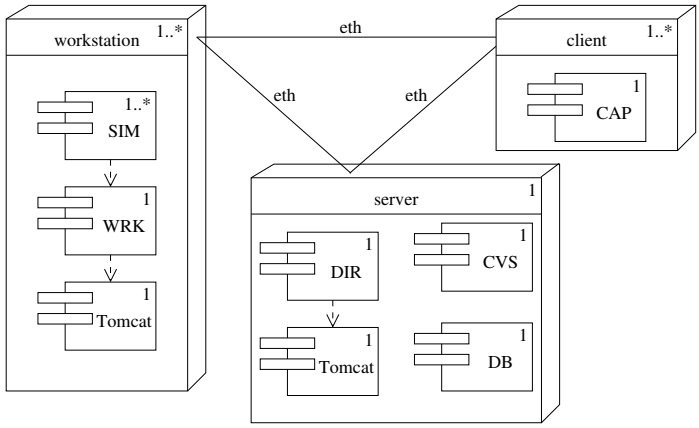

Fig. 3. The DRS architecture.

3 Realization

DRS is designed as a *grid-computing system* [18]: The simulation work should be done by standard workstations that are registered with some central information service and can be used by various clients. Figure 3 shows the global architecture: There is exactly one central server machine, one or more workstations that do the simulation computation, and one or more client machines. All these nodes are connected by some standard Ethernet network.

The server runs the central information service called DIR. This is embedded in a Tomcat server, which is a very stable and operating-system independent platform. Tomcat [2] is an open-source web server written in Java. It is designed as a runtime environment for web services and provides remote management facilities for these. It can thus be used as an application server, as we do it in DRS. The server machine usually hosts additionally a database and CVS service. The latter is used for an integrated development process that allows updating

most of DRS' components while the system is running! The DIR must always run since it is the only common service point that all system components must know.

On each workstation there is a WRK service – also running inside a Tomcat – that provides the worker's facilities to the system. The WRK registers itself with the central DIR. The workstations can be shut down, in which case the WRK unregisters with the DIR. Thus, the DIR always knows about all living workers.

The clients for the user run on some possibly smaller terminals, there may be several clients at a time. The CAP clients contact the DIR to get the list of available workers and – after the user has configured the simulation to be done – reserve there workers for their own use. When the simulation is started, the WRK servers create SIM objects – possibly more than one each – that do the local simulation. The algorithm control is either done centralized inside the CAP or decentralized within the WRKs.

Although it is still a prototype, our implementation is already very stable: The DIR usually runs for weeks without the need for restarting it, and we recently did 900 successive simulations on one and the same running WRK instances. (We usually start a CAP for every simulation, so I don't know really how many simulations one running instance could do, but we did several thousands of tests without a crashing CAP.)

4 Case Study

The major example, we are working with in the SIMONE project, is based on real world data of a part of the German railway. The following table summarizes its characteristics:

	Example	**Monday**	**Germany**
sum track length [km]	1006.503		65005
operating sites OS	104		
avg track length / OS [km]	9.678		
track sections	7111		
avg track section length [km]	0.142		
avg track sections / OS	68		
avg trains / day	795	781	34950
avg trains / OS	123		
avg OSs / train	11		
avg track sections / train	266		
avg train way / train [km]	43.509	37.844	
avg train way / day [km]	34583.981	29556.384	

Each track section is a part of the railway network, delimited by signals, train-end-detectors, switches, or the like. The example's timetable knows about

1118 trains altogether. It contains, however, data for trains running for example from Monday to Friday, only on weekends, or every day. Thus, only a part of all trains run actually on e.g. Monday. The *Germany* column – taken from [6] – gives you some impression on how the example relates to the whole German railway system.

The main parameters that specify a concrete simulation problem are: the (part of the whole) timetable (e.g. Monday), the trains (e.g. all that start between 9 and 10 a.m.), and the spatial parts of the network. For the DRS system, the user additionally can select mainly: the set of workers to be used, the partitioning, synchronized (deterministic) or non-synchronized (non-deterministic) operation, and central or de-central control.

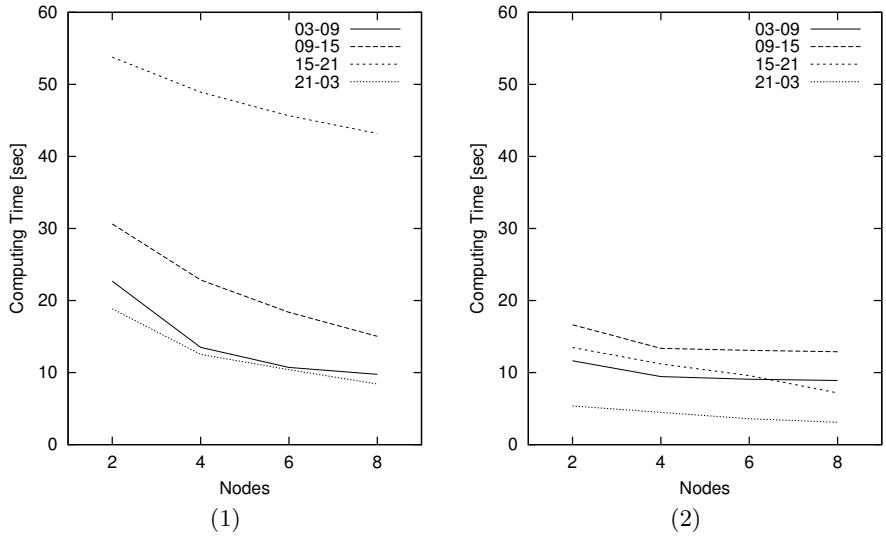

Fig. 4. Experimental results: all 104 OSs (1), and a connected selection of 67 (2).

Figure 4 shows some empirical results of DRS: In (1), we simulated the whole example, while in (2) we used only about half of the operating sites or network. In both cases, we separately simulated the trains from 3 a.m. to 9 a.m., from 9 a.m. to 3 p.m., from 3 p.m. to 9 p.m., and from 9 p.m. to 3 a.m. So, here, we did not do the whole day but different 6-hour timeslices. And, we tried all this on 2, 4, 6, and 8 computing nodes. Each node is equipped with 1GB of memory and 2 AMD K7 processors working at 1.2 GHz and runs Red Hat 8.0 Linux 2.4 and Sun's Java 1.4.0.

We can see from (1) that the computing times for simulating the whole network may differ greatly from timeslice to timeslice: The 15-21 one takes more than 50 seconds on 2 nodes and even on 8 nodes more than 40 seconds, while simulating the trains from 09-15 takes between 15 and 31 seconds. This comes

mainly from the different problems' complexity: There are 199 trains in timeslice 03-09, 243 in 09-15, 259 in 15-21, and 199 in 21-03. But although the number of trains in 09-15 and 15-21 are not too different, the latter takes about twice as long as the former. Sometimes there are local problems, that are hard to solve, e.g. when there are many trains very close together. And this is not necessarily exactly the same situation in all timeslices. So, obviously, there is some exceptional problem in the 15-21 slice.

If we take only a spatial part of the network – 67 connected OSs out of the whole 104, see Figure 4 (2) – the simulation times get reduced by an average of 50% and the exceptional problem with one of the timeslices disappears. The overall reason for these differences is that the example contains some very very heavy (or complex) OSs whose local computation takes by far longer than that of the others. And since we do not split OSs when dividing and distributing the global problem, having more computing nodes does not help solving the heavy parts. This may limit the scalability of the overall algorithm, as can be seen in the 15-21 case in (1).

We generated and compared different problem partitionings with 16, 30 and 50 parts. The 30s partitioning turned out to be best for most of the problems. So, the above tests are all based on this partitioning. I already mentioned, that the system can operate in synchronized or non-synchronized mode – here we used the synchronized mode since it always produces the same simulation result for a given problem and therefore should be preferred by most users. In fact, the non-synchronized mode is slightly faster. Additionally, we could use central or de-central control. Our implementation allows synchronized operation only in company with central control, so we used this one. We ran each particular test five times and averaged the resulting computing times.

It should be noted that solving the local simulation problem includes solving a job-shop scheduling problem. Each track section can thus be regarded as a machine that is used by *blocks* (see Section 1.2), the jobs. Each job consists of consecutive tasks. Each train uses an average of 266 track sections, thus building 266 tasks. So, in the morning timeslice 03-09, there are about $266 * 199 = 52934$ tasks, $266 * 243 = 64638$ in 09-15, $266 * 259 = 68894$ in 15-21, and about $266 * 199 = 52934$ in 21-03. And all those have to be scheduled onto 7111 machines! So, each simulation problem is quite a large job-shop-scheduling problem. DRS solves them very quickly.

5 Conclusion

I presented here the railway simulation algorithm DRS that uses – in contrast to existing approaches – constraint programming and distributed problem solving. I showed that we have a fast, stable and powerful implementation that proves the algorithms abilities empirically. Some essential features can also be testified theoretically.

This paper gives a very brief insight to my dissertation, which covers everything in much more detail. The dissertation will appear soon: [17].

References

1. A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 1993.
2. Apache Software Foundation. *Apache Tomcat*. http://jakarta.apache.org/tomcat/.
3. P. Berlandier and B. Neveu. Problem partition and solvers coordination in distributed constraint satisfaction. In *Proc. Workshop on Parallel Processing in Articial Intelligence (PPAI-95)*, Montreal, Canada, 1995.
4. K. M. Chandy and R Sherman. Space-time and simulation. In *Proc. SCS Multiconference on Distributed Simulation*, 1989.
5. Elias Silva de Oliveira. *Solving Single-Track Railway Scheduling Problem Using Constraint Programming*. PhD thesis, The University of Leeds, UK, 2001.
6. Deutsche Bahn AG. *Daten und Fakten 2002*.
7. Alois Ferscha. Parallel and distributed simulation of discrete event systems. In *Handbook of Parallel and Distributed Computing*. McGraw-Hill, 1995.
8. Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–52, 1990.
9. Markus Hannebauer. *Autonomous Dynamic Reconfiguration in Collaborative Problem Solving*. PhD thesis, Technische Universität Berlin, 2001.
10. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
11. Daniel Hürlimann. *Objektorientierte Modellierung von Infrastrukturelementen und Betriebsvorgängen im Eisenbahnwesen*. PhD thesis, Eidgenössische Technische Hochschule Zürich, 2001.
12. Volker Klahn. *Die Simulation großer Eisenbahnnetze*. PhD thesis, Universität Hannover, 1994.
13. Dirk Matzke and Maren Bolemant. Modellierung innovativer Systemtechniken der Zugbeeinflussung mit constraint-logischer Programmierung. In *ASIM – Symposium Simulationstechnik*. SCS Europe, 2003.
14. B. C. Merrifield, S. B. Richardson, and J. B. G. Roberts. Quantitative studies of discrete event simulation modelling of road traffic. In *Proc. SCS Multiconference on Distributed Simulation*, 1990.
15. Jörn Pachl. *Systemtechnik des Schienenverkehrs*. B. G. Teubner, 2000.
16. Georg Ringwelski. *Asynchrones Constraintlösen*. PhD thesis, Technische Universität Berlin, 2003.
17. Hans Schlenker. *Distributed Constraint-based Railway Simulation*. PhD thesis, Technische Universität Berlin, to appear.
18. Detlef Schoder, Kai Fischbaum, and Rene Teichmann, editors. *Peer-to-Peer*. Springer, 2002.
19. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5), September 1998.