

PADUA UNIVERSITY. ITALY
MASTER IN COMPUTER SCIENCE
COURSE IN CONCURRENT AND DISTRIBUTION SYSTEM

Discussion on the Concurrency and Distribution Issues in a Railway System Simulation

STUDENT

Alessandro Temil

SUPERVISOR

Tullio Vardanega

March 13, 2007

© 2005, by Alessandro Temil

Author's e-mail: atemil@gmail.com

Abstract

This work is the class project for the *Concurrent and Distributed Systems* course, computer science major, University of Padua (Italy) 2005-2006, professor Tullio Vardanega.

The work addresses the modeling of a railway system from a theoretical and practical standpoint, with emphasis on the concurrency and distribution issues.

For sake of simplicity, the model that I developed doesn't take into consideration possible faults, delays, bad weather or strikes that can affect a real railway system.

The main theoretical result is a system architecture that models a complex line topology, with both single and double tracks; a discretionary number of trains and passengers; railway stations with multiple platforms. It's possible to reserve a ticket for an itinerary that spawns over multiple trains: this is done preserving a large degree of concurrency without entering into inconsistent states or introducing potential deadlocks. The practical achievement is the implementation of the architecture using the Ada95 language and its distributed system annex.

Contents

1	Methodological Premise	1
2	Concurrency and Distribution Issues	3
2.1	Track	3
2.2	Platform	4
2.3	Ticket Booking	5
3	Railway System: Topology	7
4	Architecture	11
4.1	Logic Architecture	11
4.2	Concurrent Specification	11
4.3	Train	12
4.4	Passenger	13
4.5	Railway Station	14
4.5.1	Platform	14
4.5.2	Ticket office	14
4.6	How to book EuroStar trains	14
4.7	Notice Board	15
4.8	Central Control	15
4.9	Track	16
4.10	Topology	16
4.11	Distribution	16
	Conclusions	23
	Bibliography	25

1

Methodological Premise

I describe the entities that populate my model using the HRT-UML methodology. This chapter contains a brief recall of the main HRT-UML concepts. The reader not familiar with this formalism should consult [VNMD04, VZP05, BW995].

HRT-UML entities can be terminal or non-terminal and without or with a control flow. A non-terminal entity is decomposed into simpler entities until it is fully specified. There are 5 categories of entities:

- an *active* entity is an entity which lives outside the model. It has thread, and its behaviour has no interference with the model.
- a *cyclic* entity if it is made up of one non-terminating thread which has only one activation event of type *time*.
- a *sporadic* entity is made up of one non-terminating thread which has only one activation event of type *event*. Such an event has the type either software or hardware (i.e. an interrupt).
- a *protected* entity is a system resource with an access protocol with guarantee of mutual exclusion on access to the resource internals.
- a *passive* entity is a system resource with an empty access protocol.

I implement the resulting architecture using the Ada language. Despite some of its idiosyncrasies, the features of this language make it the best candidate to describe a system where concurrency plays an important role.

Where appropriate, I use UML2 diagrams made with the *Umbrella* tool to illustrate my architecture.

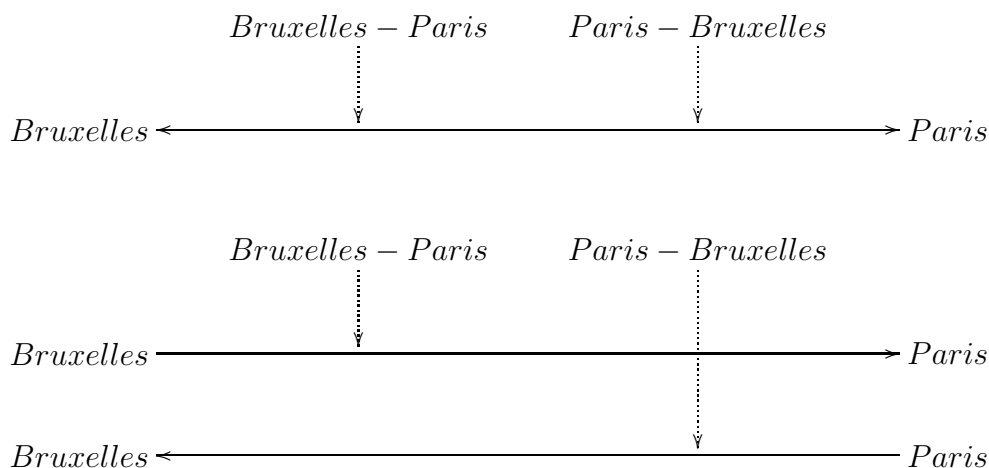
2

Concurrency and Distribution Issues

There are many issues related to concurrency that emerge from the modeling of a railway network. In this chapter I discuss some of the more intriguing ones.

2.1 Track

Tracks are shared resources among multiple trains. I define them in the model as protected entities.



As I am creating a discrete simulation, it might be plausible to describe the cruise of a train only as a set of subsequent stops. However this would not capture the improper situation exposed in the following example:

Exemple 1 (Segment Management) *Let A, B be two railway stations such that two trains go from A to B by stopping in two different platforms. If the trains leave A at the same time, then they would simultaneously and magically cover the same track at the same time. In a real model with a unique leg, this situation is an absurd. To make the simulation realistic, the trains must travel one after the other.*

In my discrete simulation, a train being traversing a track is part of the simulated state. Moreover, I could line more trains up the same track if their start is reasonably spaced out and their speed preserves the safe distance (for the safety of the passengers).

Protocol for the Controlled Concurrency Management. Without loss in generality, I discuss a case-study with only two trains. If the trains have different types, i.e. ES and IR, then ES train goes first since its priority is more high than IR. If the trains have same types, i.e. either ES or IR, then the train with the free arrival-platform goes first. Otherwise, the train with a lesser identifying number goes first.

Simplification. For the sake of simplicity, I choose to model only one train for one track at a time.

2.2 Platform

A number of intriguing subtleties and problems arises when modeling the platforms. What happens when a train arrives and its designated platform is occupied? I never took into serious consideration to create a platform for each train passing for a given station, as this would be too simplistic.

I assume stations with dead end tracks where trains can sit and wait for their arrival platform when busy, without keeping the transit track busy. Furthermore, I can achieve a more complex model by dropping the dead-end track requirement for all railway stations. For example, it can be reasonable for little railways stations not to have this maneuvering space. For the sake of simplicity, I introduce dead-end tracks for all railway stations. Like the segment management, the queue for the platforms depends on two factors: arrival time of the train and its priority.

As I decided to allow more trains passing for one platform, there is the possibility of passengers waiting at the same platform for different trains. A simple solution to this problem could be to wake up all the waiting passengers and let them choose whether to board or not. This obviously caused unnecessary overhead on the system. The better solution that I choose is to create a family of wake up events, one for each train that uses a given platform. This allows to selectively wake up just the passengers that have to step on the current train.

I choose to represent routes that can include multiple legs on multiple trains. This poses the problem whether to create a single queue for both the passengers that have to exit the station and for those who just need to change train or to make two different queues. Assuming the first solution, the passenger wishing to change train would have to requeue himself on the correct platform. With the second one, the control would never be returned to the passenger routine, but the platform code would take care of redirecting the passenger to the correct queue for waiting the next train in his route. I choose the second solution.

Last but not least, passengers could face a starvation problem.

Example 2 (Starvation) *Let us consider the case where, at an intermediate station, a passenger waiting for a train that is always full. That passenger would wait forever.*

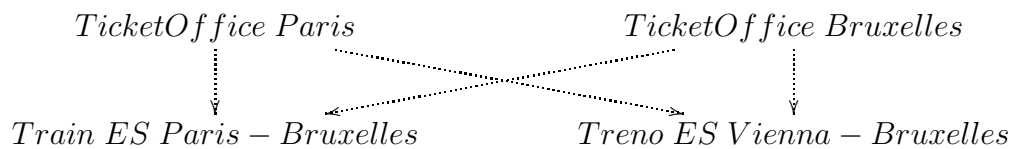
This situation is plausible also for a real railway system. I don't consider it is worth solving it in an algorithmic way as it is possible to avoid it completely by using the proper means. If a passenger needs to have a service guarantee, he could book the train in advance. A correct statical sizing of the system could also avoid starvation for the occasional traveler (in other words, adding more trains can structurally solve the problem).

2.3 Ticket Booking

Modeling the ticket purchase system required taking several design decision, whose effects were not so evident in advance. As I choose to support routes spanning over different types of train (bookable and not), it is implied that one single ticket office must support both kind of purchases (I cannot separate passengers in two separate queues).

The most interesting problem arises when we design the booking database system. Consider the following example:

Example 3 (deadlock in booking) *Passenger P_1 wants to book leg A and B, passenger P_2 wants the same legs. P_1 books A, then it is preempted and P_2 executes. P_2 successfully books B and then blocks waiting for A to become available. P_1 is also blocked for B to become available.*

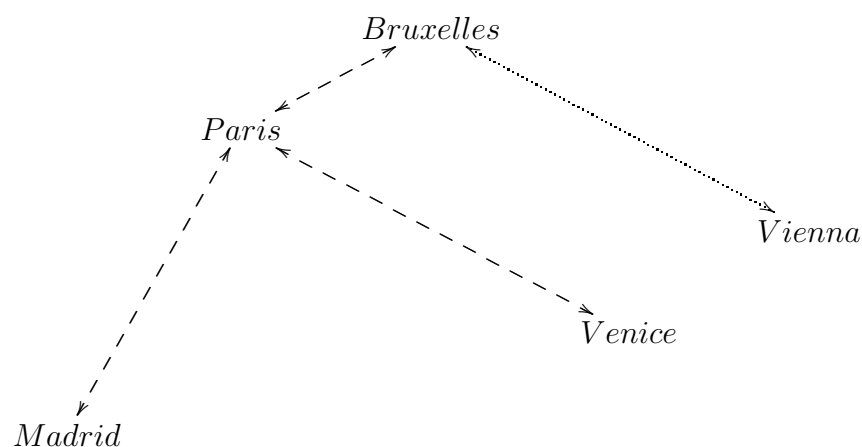


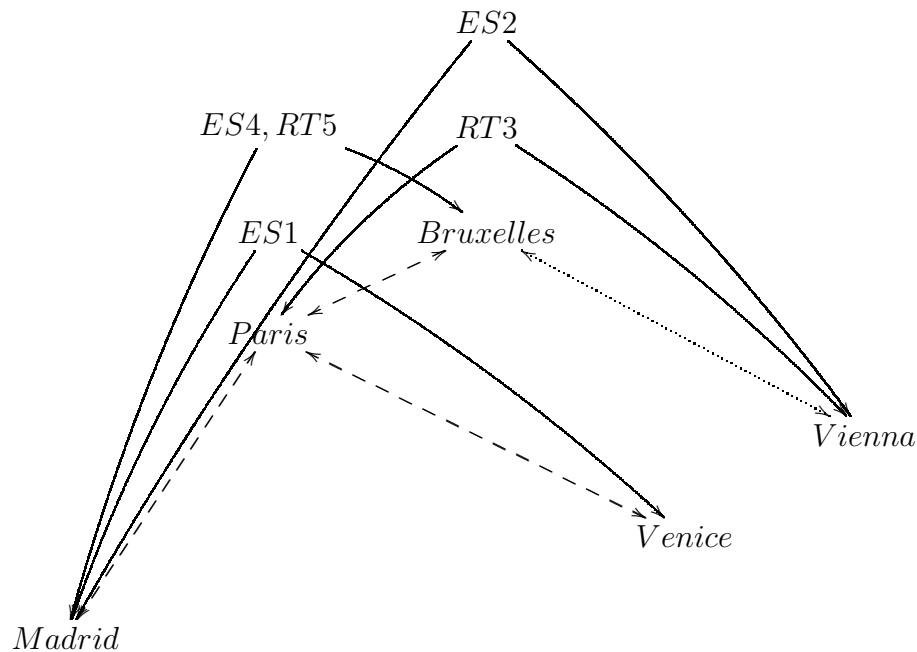
This is a well known deadlock situation. Imposing a global lock on the resources might naively solve the problem but with important drawbacks on scalability. Having P_2 resign B when it sees that it cannot book A might be another solution but it is rather unelegant and potentially unnecessary. As I discuss in Chapter 4, my solutions avoids both such issues with a clever design.

3

Railway System: Topology

In this chapter I summarize the sample railway system topology that I have used in the program. This should help the reader understanding the declarative part of the code and eventually adding new stations/trains. I use five cities Bruxelles, Madrid, Paris, Venice and Vienna, and some hypothetical railway segments that connect them. In the figure, the broken line between two cities represents a double track segment, whereas the dot line a single segment; that is, the Bruxelles-Vienna and Vienna-Bruxelles trains rase for the same track.





In table 3, I introduce the trains in my system. There are three eurostar trains (ES) and two regular trains (RT).

In table 3.2, I introduce the passengers. There are eight passengers. They can change trains, type of trains, route. For example, Gyro Gearloose takes two ES and one RT from Venice to Vienna; Pluto the Pup takes an ES from Madrid to Bruxelles, and a RT from Bruxelles to Madrid; Minnie Mouse changes her route. She goes from Vienna to Madrid by exchanging into Paris, and when she comes back, she changes into Bruxelles.

Id Train	Train	type	Track = platform + segment	Id Track
1	Madrid Venice	ES	Madrid 1, Madrid-Paris	1
			Paris 2, Paris-Venice	2
			Venice 1, Venice-Paris	3
			Paris 1, Paris-Madrid	4
2	Vienna Madrid	ES	Vienna 1, Vienna-Bruxelles	1
			Bruxelles 1, Bruxelles-Paris	2
			Paris 1, Paris-Madrid	3
			Madrid 1, Madrid-Paris	4
			Paris 1, Paris-Bruxelles	5
			Bruxelles 1, Bruxelles-Vienna	6
3	Vienna Paris	RT	Vienna 1, Vienna-Bruxelles	1
			Bruxelles 1, Bruxelles-Paris	2
			Paris 1, Paris-Bruxelles	3
			Bruxelles 1, Bruxelles-Vienna	4
4	Bruxelles Madrid	ES	Bruxelles 1, Bruxelles-Paris	1
			Paris 1, Paris-Madrid	2
			Madrid 1, Madrid-Paris	3
			Paris 1, Paris-Bruxelles	4
5	Bruxelles Madrid	RT	Bruxelles 1, Bruxelles-Paris	1
			Paris 1, Paris-Madrid	2
			Madrid 1, Madrid-Paris	3
			Paris 1, Paris-Bruxelles	4

Figure 3.1: Trains

Name	Forward and Back	From - To	Route	How
Scrooge McDuck	Forward	Vienna - Madrid	Vienna - Bruxelles Bruxelles - Madrid	RT RT
	Back	Madrid - Vienna	Madrid - Bruxelles Bruxelles - Vienna	RT RT
Goofy Goose	Forward	Venice - Paris		ES
	Back	Paris - Venice		ES
Daisy Duck	Forward	Paris - Madrid		ES
	Back	Madrid - Paris		ES
Minnie Mouse	Forward	Vienna - Madrid	Vienna - Paris Paris - Madrid	RT RT
	Back	Madrid - Vienna	Madrid - Bruxelles Bruxelles - Vienna	RT RT
Gyro Gearloose	Forward	Venice - Vienna	Venice - Paris Paris - Bruxelles Bruxelles - Vienna	ES RT ES
	Back	Vienna - Venice	Vienna - Bruxelles Bruxelles - Paris Paris - Venice	ES RT ES
Pluto the Pup	Forward	Madrid - Bruxelles		ES
	Back	Bruxelles - Madrid		RT
Donald Dug	Forward	Bruxelles - Venice	Bruxelles - Paris Paris - Venice	ES ES
	Back	Venice - Bruxelles	Venice - Paris Paris - Bruxelles	ES ES
Mickey Mouse	Forward	Bruxelles - Madrid		RT
	Back	Madrid - Bruxelles		RT

Figure 3.2: Passangers

4

Architecture

In this chapter, I focus on the architecture of the simulation model. Where appropriate, I use UML2 diagrams to illustrate the relationships that link the different entities. For the sake of simplicity, there we assume there is no fault model to consider in this architecture; that is to say: *“il modello è infallibile. È la realtà, spesso, a essere inesatta.”*

4.1 Logic Architecture

In diagram 4.1, I sketch the logic architecture of the model. Each station has its own set of platforms, a ticket office and a notice board. Every platform is responsible for updating the local notice board as well as the central control. The ticket office deals with a set of ticket databases, one for each train that permits booking. Each train has its route, which is a set of stages. Each stage is a couple “departure platform - segment” and it is implied that the arrival platform is the departure platform of the next stage. Each segment can refer to a single (bidirectional) or to a double track. Each traveler is also statically defined with by route.

4.2 Concurrent Specification

In diagram 4.2, I focused more on the concurrent aspects of the architectural model. The *traveler* active entity purchases a ticket at the ticket office and is enqueued at the departure platform. A detailed analysis of this part will be discussed extensively in diagram 4.3. The *train* active entity arrives at the platform, wakes up the travelers that have to step down at that stop, then wakes up the travelers that have to step on from the platform, then rideparts. The platform notifies the events to the global “central control” and to the local “notice board”. When departed, the train races with

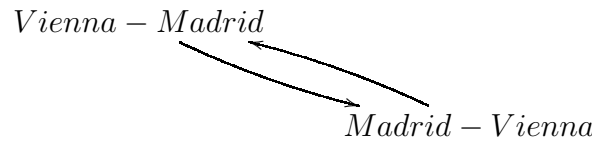
other trains for the *track* resource. The *track* is also responsible for notifying the central control of the trains that pass over it.

In the next paragraphs, I detail on the various entities using the HRT-UML model.

4.3 Train

The train is defined as a *cyclic* entity that periodically traverses its route. The train is suited to be represented by a cyclic object as it has its own control flow and executes a looping activity, as we can easily see from the next example.

Example 4 (Train) *The train Madrid-Vienna cyclically executes the following route: Vienna-Bruxelles, Bruxelles-Paris, Paris-Madrid, Madrid-Paris, Paris- Bruxelles, Bruxelles-Vienna, and so on.*



I defined two typologies of train: The Eurostar and the local train. Eurostar has mandatory booking of the seats, whereas local train does not require booking. I didn't model trains with non-mandatory booking as they would have added code complexity without adding much value to the simulation. Eurostar have a higher task priority than local trains so they have precedence in acquiring resources (tracks and platforms).

Definition 1 (Stage) *Stage is an ordered couple of the departure platform and the railway segment that is traversed.*

Example 5 (Stage) $\langle \text{Venice platform 2}, \text{Venice-Paris} \rangle$ is a stage.

Definition 2 (Route) *The route of the train is an ordered list of stages. The stages must form a circular loop.*

Example 6 (Route)

$$ES \text{ Bruxelles-Madrid} = \langle \langle 1, \text{Bruxelles-Paris} \rangle, \langle 1, \text{Paris-Madrid} \rangle, \langle 2, \text{Madrid-Paris} \rangle, \langle 2, \text{Paris-Bruxelles} \rangle \rangle$$

Each stage is defined as $\text{stage} = \langle \text{departure platform}, \text{segment} \rangle$. The arrival platform for each stage is the departure platform of the next stage. The arrival platform for the last stage is the departure platform for the first.

A train is defined by the tuple containing its name, its numeric id, its category, the route it follows, the number of seats and a link for the booking database (where appropriate). In the case of an Eurostar, the number of seats is used also evaluate booking requests, whereas in the case of a regular train, it is just used as the maximum number of passenger that can be on board at the same time.

```
ES Bruxelles-Madrid = ( (1, Bruxelles-Paris), (1, Paris-Madrid),
                        (2, Madrid-Paris), (2, Paris-Bruxelles) ), 20places
```

4.4 Passenger

In diagram 4.3, I represented a detail of the concurrent specification focusing on the actions that occur from the passenger side. The passenger shows up at the ticket office of his departure station. If the ticket that he wants doesn't require booking, it is immediately transferred at the departure platform. Otherwise, the ticket databases of the trains involved in the route have to be accessed. This can be done concurrently with other passengers booking the same train thanks to the approach described later. After all the legs have been booked, the passenger is transferred to the departure platform.

From a concurrent point of view, I can categorize the passenger as a *sporadic* entity. Its activation event is the departure and the arrival of the trains it uses to travel.

In my architectural definition, the passenger always accesses the station from the ticket office (s/he never goes directly to the platforms). S/he cannot purchase a round-trip ticket. S/he is forced to buy just the ticket s/he'll use for the current trip. Such a situation was chosen to keep the code complexity low, it is not caused by any logical limit in the model.

I map "active" entities to Ada'95 Tasks, typed channel are implemented as `entries` (when guarded) or `procedures` (when not guarded). By using the powerful **entry family** construct, I was able to parametrize access channels to the platforms over the number of the trains passing through it.

```

entry depart(for T in Range_Trains)
    (legs : Legs_Sequence_Ref_T;
     travel_index : Integer_Ref_T;
     passengername : String_Ref_T) when
        trainboarding /= null and
        number_train_stopped = T and tFree_Seats > 0 is

```

4.5 Railway Station

A station is defined as the aggregate of the ticket office, the local platforms and the notice board. None of these entities has its own control flow, and they are all accessed in mutual exclusion.

Station = ticket office + platforms + notice board

4.5.1 Platform

The platform is defined as a *protected* entity as it doesn't have a control flow and has an access protocol that depends on its internal state. It is a local resource and it is accessed in mutual exclusion.

The platform is accessed by the trains and by the passengers. The passengers are queued to wait a specific event (arrival or departure of a train), the trains can change the state of the platform and thus have the passenger board or descent.

4.5.2 Ticket office

The ticket office is defined as a *protected* entity as it is accessed by passenger in mutual exclusion.

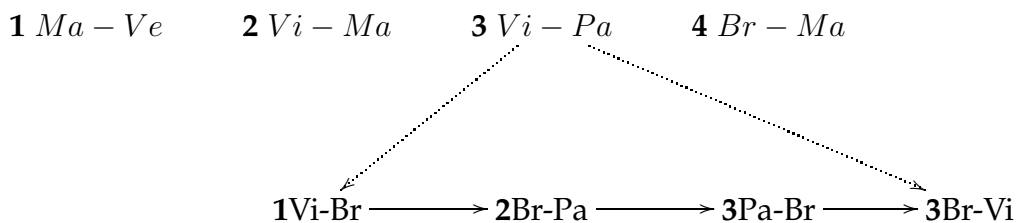
4.6 How to book EuroStar trains

Booking tickets for itineraries spanning over multiple trains is an activity naturally subjected to deadlock. The naive solution, to lock the entire database, is rather unellegant and definitely not viable if we want to be able to scale the system to support a large number of users. It is known that forcing the resources to be always acquired in a given order is one of the ways to eliminate the risk of deadlock. I used such a

knowledge to model my system for permitting concurrent booking without the risk of deadlock or inconsistency. Every stage of a route is booked separately, but following a superimposed order. In the implementation, there is one different protected resource for each train, and this means that the user booking a leg on one train prevents other users from booking on the same train. However, it is easy to see how locking could easily be implemented at stage level, obtaining the maximum locking granularity.

A ticket database entity is a *protected* entity. One is created for each train.

Let us consider the following diagram.



Let us suppose that Scrooge McDuck wants to book a ticket from Bruxelles to Venice. Scrooge McDuck wants the Bruxelles-Madrid Eurostar from Bruxelles to Paris and the Madrid-Venice Eurostar from Paris to Venice. As the Madrid-Venice Eurostar is train number 1, the booking system starts from booking the appropriate segments on such train, even if it is not the first leg of the Scrooge McDuck route. So the segments are booked in the following order: 1.2 (Madrid-Venice Eurostar, Paris-Venice stage) 4.1 (Bruxelles-Madrid Eurostar, Bruxelles-Paris stage).

4.7 Notice Board

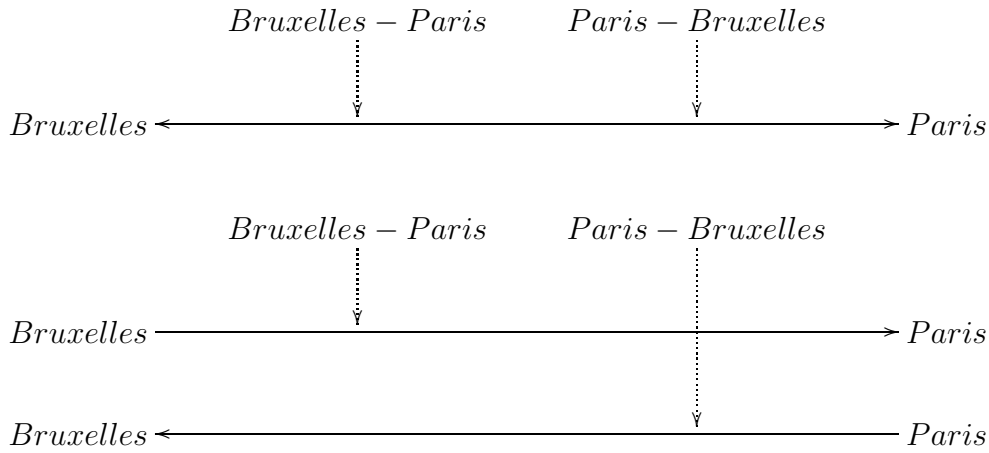
The notice board is a *protected* entity. there is a notice board for each station, that shows arriving and departing trains. The platform are responsible for notifying these events to the notice board.

4.8 Central Control

The Central Control is a *protected* entity. It shows the current status of all the trains and passengers active in the system. It receives the informations from the platforms and the tracks.

4.9 Track

The Track is a *protected* resource, shared among the trains. I used both single and double tracks. A single track is used in both directions, whereas for a double track, each line is used in only one direction.



I modeled my architecture to sit on a system with *Immediate Priority Ceiling* (IPC). Processes representing EuroStar trains are given a higher priority so they have precedence in acquiring tracks and platform.

4.10 Topology

The topology of the model is an oriented graph, where nodes are the stations, the arcs are the tracks and their weight is the length of the track. I use a complete topology, in other words, if there is a Paris-Madrid segment, there is also a Madrid-Paris one. The system works also if the topology is incomplete.

4.11 Distribution

In diagram 4.4, I show the distribution aspects of the architecture. I implement the whole simulation (trains, stations, tickets, etc) in one node, whereas the views (central control and notice board) can sit on separate nodes.

I consider three possible ways to distribute the program, I discuss them here:

1. The first and most intuitive idea is to distribute stations among different nodes. This would imply that trains (processes) have to move. Moving code, together with its execution context, is a complicated thing to do. There are a number of other reasons why this would be a bad choice, but the difficulty of implementing movable code is more than enough to discard this solution.
2. Another idea is to distribute the resources (stations, platforms, tracks) on one node and the processes (trains, passengers) on another. With this solution, I would transmit requests to acquire and release resources. Such a situation introduces a large degree of nondeterminism, as those requests have to travel over a network so their order would be hard to maintain. As I want to simulate the concurrent aspects of the system, this solution is not very attractive to me.
3. The third possibility is to separate the model (that in this case is the whole simulation) from the views (the central control and the notice boards) as for the MVC (Model View Controller) *Pattern*. As the communication is mostly unidirectional, I chose to use asynchronous communication. This involves the tradeoff that the view might not resemble the exact state of the resource at a given time. If it was not acceptable, we would only need to switch to synchronous communication with *at least once* semantics, as the repetition of a message would not cause problems.

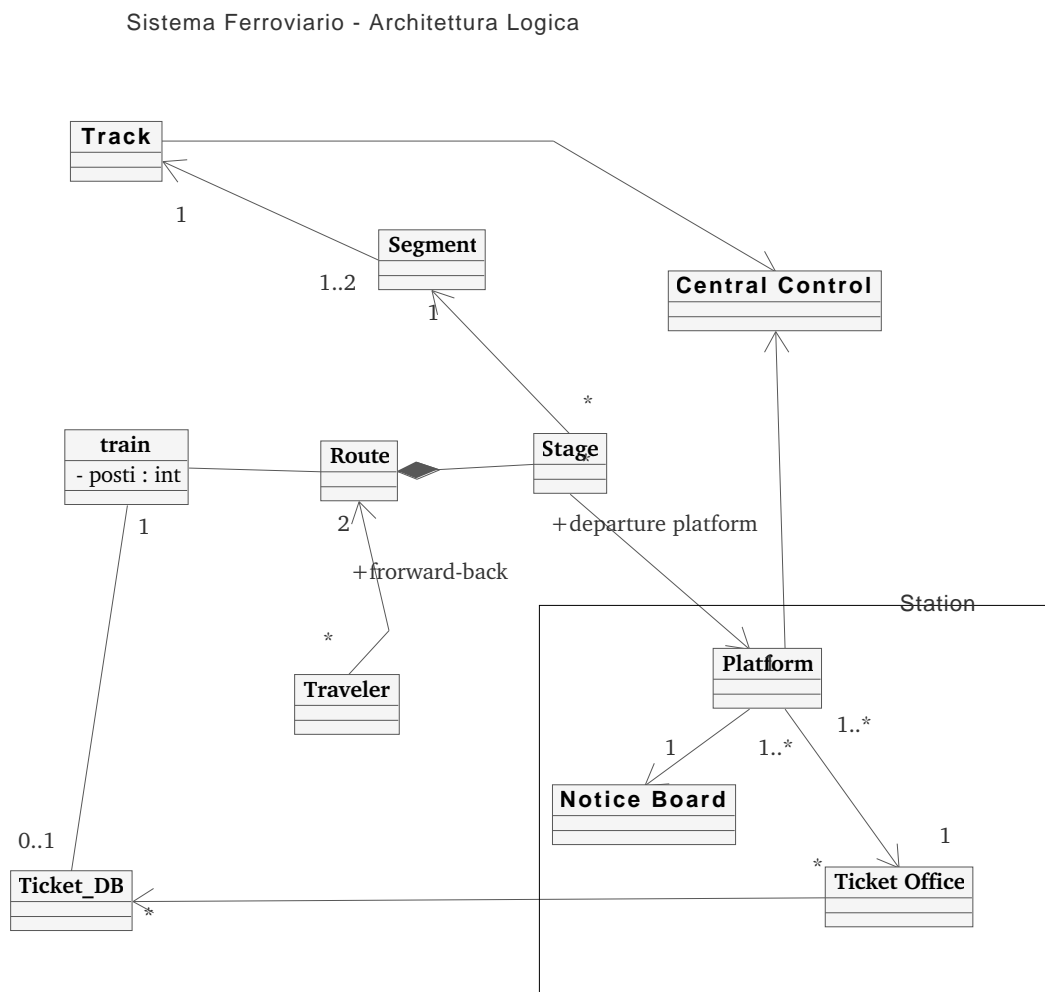


Figure 4.1: Logic architecture in UML2

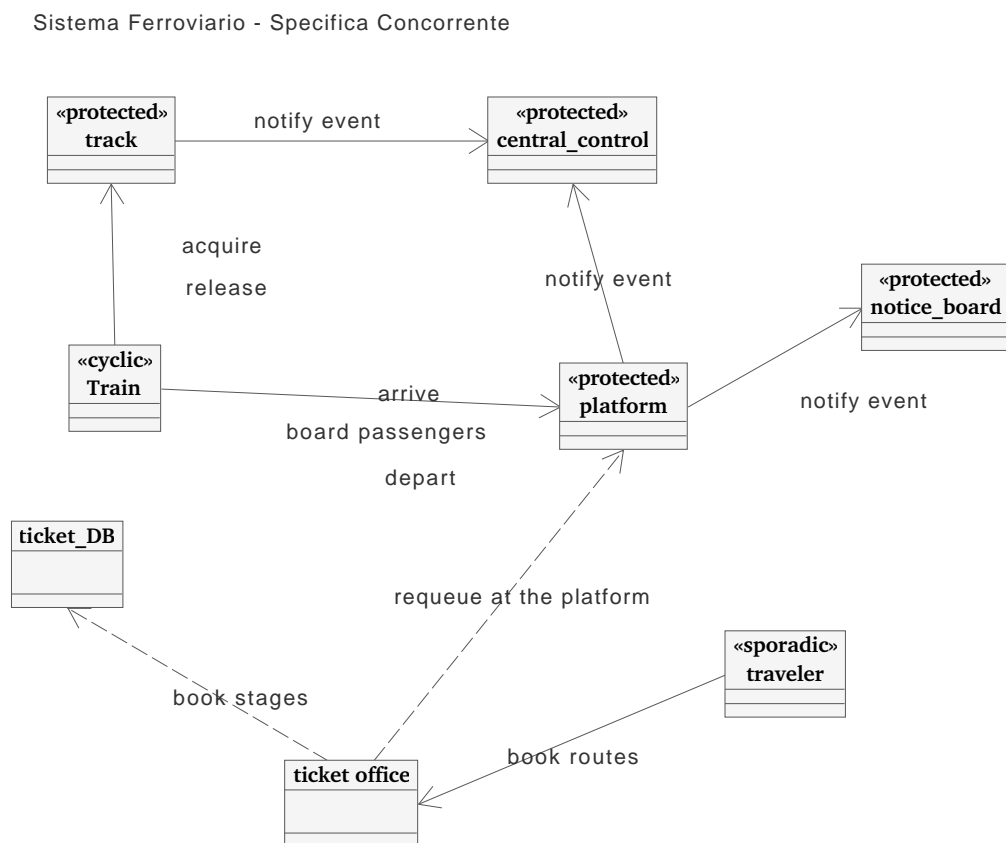


Figure 4.2: Global Concurrent Specification

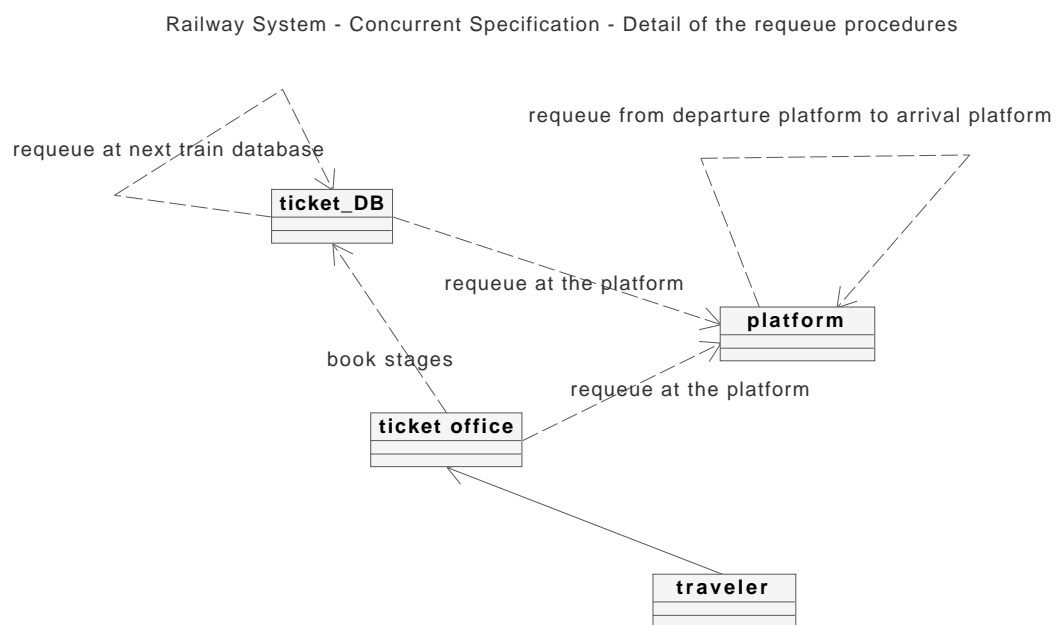


Figure 4.3: Passenger Concurrent Specification

Railway System - Distribution

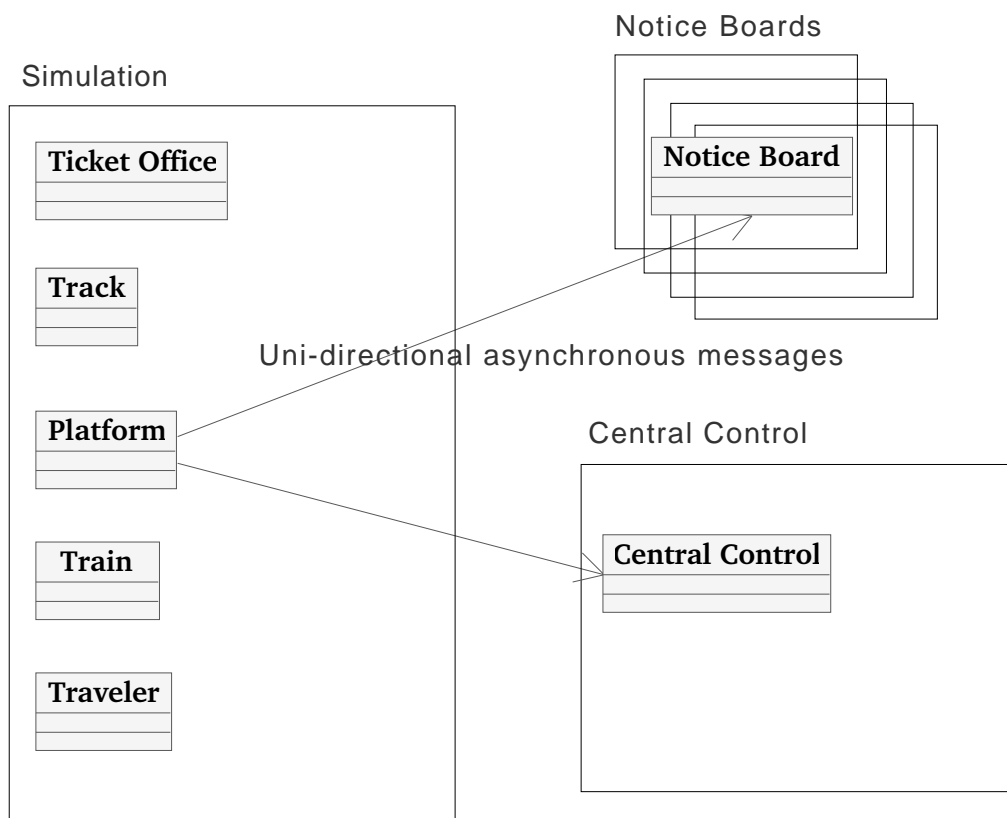


Figure 4.4: Distribution Specification

Conclusions

I defined a concurrent architecture the models a railway system, and implemented it using the Ada programming language. An important achievement is the algorithmic solution for the concurrent booking of complex routes. The proposed algorithm handles concurrent booking by multiple users with a fine grained locking policy, and completely avoids deadlock and inconsistent states. An significant application result is the selective wake up of the awaiting passengers by using the `entry family` language construct. With a clever exploitation of the language features I achieve the lowest possible overhead on the system without sacrificing the clarity and brevity of the code.

Future works

This realization can be further expanded in at least two directions. First, faults could be introduced to obtain a richer simulation model. Furthermore, the architecture could be completely encoded in an HRT-UML design to take advantage of an automatic code generation facility, as described in the paper by Bordin e Vardanega *A New Strategy for the HRT-HOOD to Ada Mapping* [BV05].

Acknowledgements

I would like to express my gratitude to Tullio Vardanega for setting challenging quality requirements that have encouraged me to look for intriguing solutions.

A special thanks goes to Daniela Cancila for the fruitful discussions over the architectural topics and for carefully reviewing the drafts of this document.

I am grateful to Marco Comini for the latex style used for this paper.

Bibliography

- [Bar98] J. Barnes. *Programming in Ada 95*. Addison Wesley, seconda edizione, 1998.
- [Bar05] J. Barnes. Rationale for Ada 2005: 3 Structure and visibility. www.adacore.com, visitato in dicembre 2005.
- [BV05] M. Bordin and T. Vardanega. A New Strategy for the HRT-HOOD to Ada Mapping. *Int. Conf. on Reliable Software Technologies Ada-Europe 2005*, Springer-Science, LNCS(3555):51–66, 2005.
- [BW00] A. Burns and A. Welling. *Concurrency in Ada*. Cambridge University Press, seconda edizione, 2000.
- [BW995] *HRT-HOOD: A Structural Design Method for Hard real-time Systems*. University of York (UK), Elsevier, 1995.
- [Gro05] Object Management Group. Uml Resource Page. www.uml.org, Ultima visita in Dicembre 2005.
- [VNMD04] T. Vardanega, M. Di Natale, S. Mazzini, and M. D'Alessandro. Component-Based Real-Time Design: Mapping HRT-HOOD to UML. *Proc. 30th Euromicro Conference*, IEEE CS Press:6–13, 2004.
- [VZP05] T. Vardanega, J. Zamorano, and J.A. De La Puente. On the Dynamic Semantics and the Timing Behaviour of Revanscar Kernels. *Real-Time Systems*, Springer-Science, 29:58–89, 2005.