Using the Tools in TRADE III: A Controller for a Compact Dynamic Bus Station

R.J.Wieringa[1]

July 21, 1999

[1]Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands. Email: roelw@cs.utwente.nl, http://www.cs.utwente.nl/∼roelw

Contents

Chapter 1

Introduction

TRADE is a set of Techniques for Requirements and Design Engineering of software systems. It was defined in the course of analyzing nineteen object-oriented and six structured software specification and design methods [5, 4]. The techniques in TRADE are applied here to the specification of a compact dynamic bus station, which is a bus station in which busses are dynamically allocated to platforms. Dynamic allocation can save space in comparison to static allocation, because in dynamic allocation, platform space need not be reserved for busses that are not present. The TRADE specification is based upon a prototype made by Klusener *et al.* [1]. This report is intended as an illustration of the ideas in TRADE. In two earlier case studies, a TRADE analysis was made of a decision support system for traffic maintenance [2] and of a meeting scheduler system [3].

A compact dynamic bus station consists of a number of platforms, sensors and information screens. There are sensors in the approaching roads, that can sense whether a bus is approaching. Sensors at the entries and exists of the station tell whether busses are entering or departing the bus station, and sensors at the platforms can sense whether busses are parked at the platform. One platform is reserved for busses that, for some reason, could not park at their intended platform. This platform is called the buffer. Each bus contains a screen upon which messages to the driver can be displayed. This is called a driver instruction screen. The bus station has several passenger information screens upon which information for the the passengers can be displayed. Busses drive trips, and each trip has a route that is called a line. Each day, one or more busses can make several trips of one line. There is a database that stores the reservation of a platform for a particular line. The required system should, as much as possible allocate an incoming bus to the platform reserved for the trip that the bus is now making. There is a linear programming package called CPLEX that can compute optimal allocations. When a bus enters the station, the driver should be instructed to drive to the platform allocated to the trip and the passengers should be informed of this allocation. More detailed requirements are given below.

Chapter 2

Business Requirements

The social system in which the bus allocation system is embedded consists of passengers, bus drivers, the bus company, and local and national governments. A number of business requirements originate from these stakeholders, such as ease of understanding of the system output, timeliness of the output, legal requirements, cost requirements for the owner of the station, etc. These are not detailed here.

Chapter 3

System Requirements

3.1 System Definition

The **Bus Allocation System** (BAS) is a software system that monitors bus trips, allocates busses to platforms and informs drivers and passengers of this.

3.2 Mission

Mission:

- Dynamically allocate busses that approach a compact dynamic bus station (CDB) to platforms.

Responsibilities:

- Monitor the state of a bus trip.
- Periodically recompute the optimal allocation of trips to platforms.
- Announce allocations to drivers in approaching busses and to passengers waiting at platforms.

Exclusions:

- The system does not handle *reservation* of platforms for busses; rather, given a set of reservations, it will dynamically compute an allocation of busses to platforms that is optimal with respect to these reservations.
- The system does not deal with linked trips (e.g. those for which passengers of one trip must be able to transfer to another trip).

3.3 Function Refinement

Figure 3.1 represents the desired system functions as a refinement of the system mission. It relates the system functions to the mission and responsibilities listed in section 3.2. The abbreviation "CRUD" stands for Create, Read, Update or Delete. These are the standard database functions to create and maintain data.

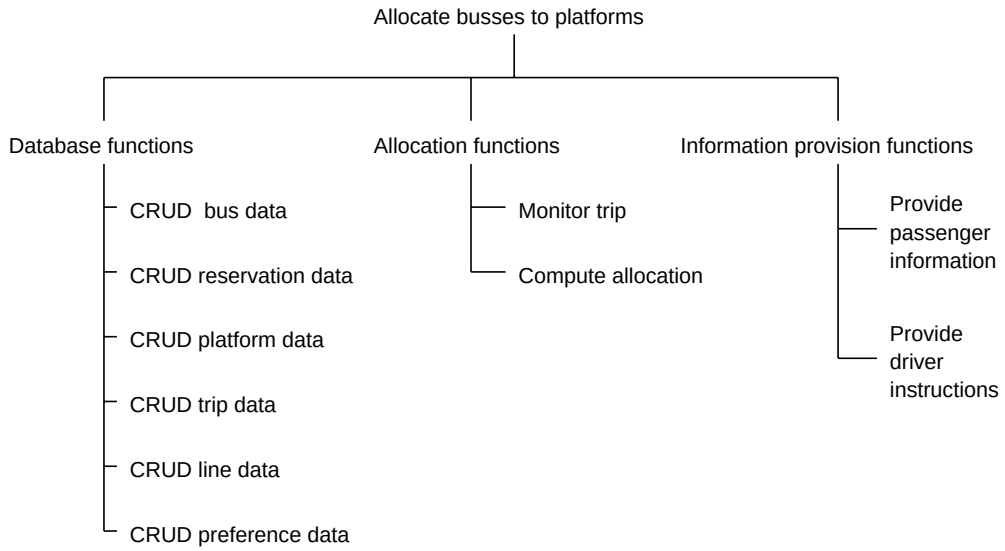

Figure 3.1: Function refinement tree.

3.4 Context Model: Classification and Multiplicity

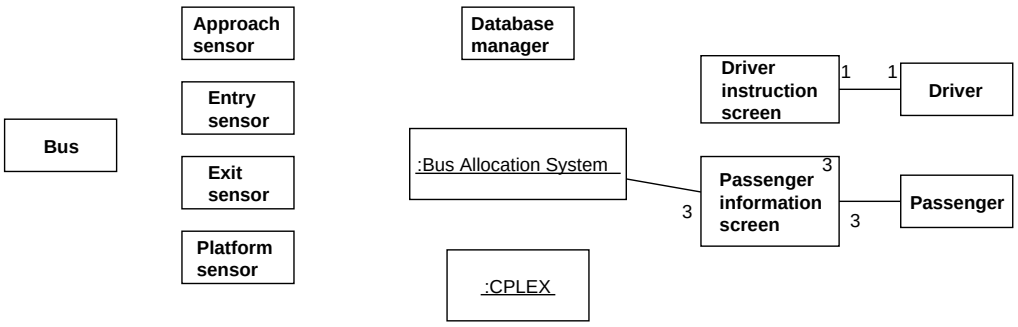

Figure 3.2: Classification and multiplicity of entities in the context of the system.

Figure 3.2 is a class diagram showing the system and the (classes of) entities in its environment. Lines are included in the class diagram of the context only if needed to indicate multiplicities. The multiplicities are snapshot multiplicities, i.e. they are properties of the number of entities that can exist simultaneously at any point in time (rather than over a period of history). Figure 3.2 shows the classification and multiplicity of entities in the context of the system. There are three passenger information screens and an arbitrary number of other classes of entities.

3.5 Context model: Communication

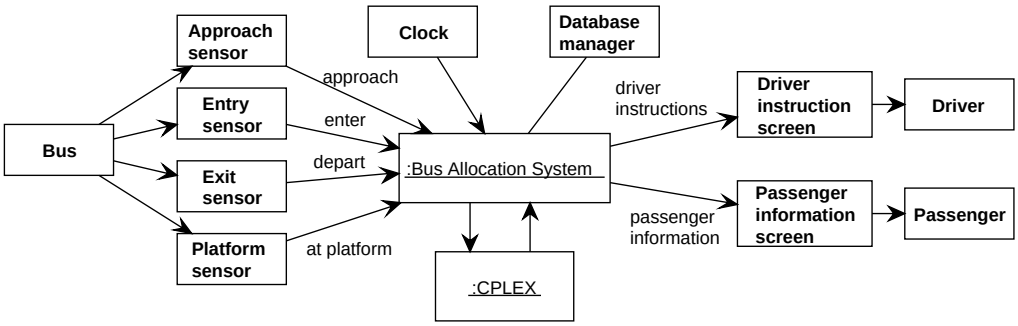

Figure 3.3: External communications.

Figure 3.3 shows the communication channels in the context of the system.

3.6 System Behavior: Event List

Figure 3.4 lists the events to which the system must respond, the sensor that observes these events, the stimulus sent by the sensor to the system, the desired response of the system and the functions to which each event–response pair contributes. This relates the event list to the function refinement tree. The sensors mentioned in this list must appear in the communication model of the context, and conversely the sensors listed there, must be used in the event list.

Event	Source	Sensor	Stimulus	Desired response	Function
Bus approaches	Bus	Approach sensor	approach	Update passenger information board	Monitor trip Provide passenger information
Bus enters bus station	Bus	Entry sensor	entry	Update passenger information board. Instruct driver to drive to platform or buffer	Monitor trip Provide passenger information Provide driver instructions
Bus arrives at platform	Bus	Platform sensor	at platform	Clear driver instructions	Monitor trip Provide driver instructions
Bus departs from bus station	Bus	Exit sensor	exit	Clear passenger information board Recompute position assignment	Monitor trip Provide passenger information
Bus expected	Time	Clock	Scheduled arrival 15 minutes from now	Update trip status to reserved2	Monitor trip
Bus due	Time	Clock	Scheduled arrival time is now	Trip status becomes delayed	Monitor trip Provide passenger information
Time to compute allocation	Time	Clock	Minute has passed since last computation	Compute optimal allocation	Compute allocation
Decision to update database	Database manager	Event in subject domain	update request	Update the database	All database functions

Figure 3.4: Events to which the system must react, and their desired responses.

3.7 System Behavior: State Transition Diagram

Figure 3.5 shows more of the required behavior of the system. To understand the diagram, refer to the model of the subject domain presented in appendix A.

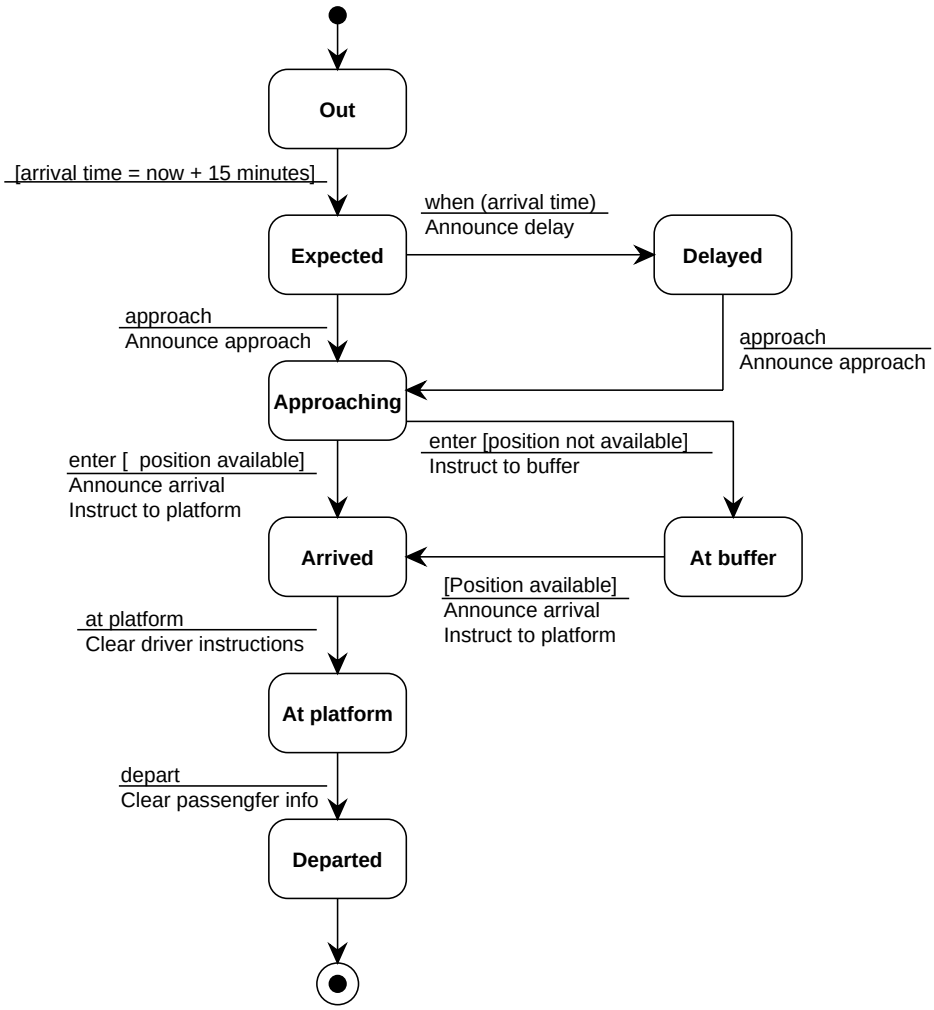

Figure 3.5: Mealy diagram of the desired response of the system to changes in trip status.

The response requirements shown in figure 3.5 assume that the bus trip behaves in an ideal manner, as shown in figure A.2. If a trip actually behaves in a different manner, the system cannot deal with that.

Property	Always true in subject domain	Violation must be registered by BAS	BAS must take corrective action when violation is registered
1. The total length of busses at platform never exceeds the length of the platform. *For all existing p ∈ Platform ,* $(\sum_{a \in p.PositionAssignment} a.Reservation.Bus.length) \leq p.length$	No	Yes	No
2. The total length of busses at platform never exceeds the length of the platform plus the maximum length of a bus.	Yes		
3. For each reservation, the starting time interval is disjoint from the departure interval. *For all r ∈ Reservation, define r. end_time = r.starting_time + r.duration. Then [r. starting_time, r.starting_time + 2 minutes] ∩ [r. end_time, r.end_time + 2 minutes] = ∅.*	No	No	
4. For different reservations allocated to one platform, their departure and arrival intervals must be disjoint.	No	No	
5. For any pair of reservations allocated to the same platform, the ordering of their departure intervals equals the ordering of their arrival intervals.	No	No	
6. The departure time of a trip is one of the departure times of the line of the trip. *For all t ∈ Trip, t. departure_time ∈ t.Line.departure_schedule, where we assume that the departure schedule is a list of departure times.*	Yes		
7. The halting time of a reservation does not exceed the reservation duration.	No	No	
8. The end time of a position assignment lies between the starting time of its reservation and the end time (= starting time + duration) of its duration.	No	No	
9. For every line-platform combination there is at least one preference.	No	Yes	Yes
10. For every line, there is at least one preference with weight 0.	No	Yes	Yes
11. Initially, the platform assignment assigns a trip to the preferred platform for the line of this trip.	No	No	
12. A bus cannot be at a buffer and a platform at the same time.	Yes		
13. A reservation can only have a position assignment to a position at the platform to which the reservation is assigned.	No	No	

Figure 3.6: Requirements on the system database.

3.8 Requirements Implied by Subject Domain Properties

The system must store data about the subject domain, and the subject domain has certain properties that lead to certain further requirements on the system. Figure 3.6 lists these properties. All properties are described in natural language. For illustrative purposes, some are expressed in simple mathematics too. The table lists a number of subject domain properties, and for each property gives the following information:

- Column 2 says whether the property can be violated by the subject domain or not. If the property is always true in the subject domain, then there cannot be violations in the subject domain; and therefore it makes no sense to require the system to register violations (the 3rd column) nor to ask the system to enforce compliance with the property (the 4th column). Rather, if the property is always true in the domain, then this entails the following requirement on the system:

  > The system must not contain data that represents a violation of this property.

  An example is that a bus cannot be at a platform and a buffer at the same time. Because this is always true in the subject domain, it is a requirement, i.e. a norm for the system. Using database terminology, it is an *integrity constraint* on the data in the system.

- If the property can be violated in the subject domain, then the next thing to decide is wether the violation must be registered by the system. For example, it is a norm for the subject domain that the total length of the busses at a platform does not exceed the length of the platform. It is very easy to violate this norm—just park an extra bus at the platform. This violation must be registered by the system, and the system is not required to do anything about it—nor indeed can it do anything about it.

  If, on the other hand, a violation must *not* be registered by the system, then again we have a case of a integrity constraint on the data in the system. For example, the starting time interval and the departure time interval of a reservation must be disjoint (property 3). The system is required not to store any data that violate this property; therefore, this is an integrity constraint of the system, even though it is also a constraint—not a truth—for the subject domain. Planners may very well make a schedule in which this constraint is violated. But because the system is required not to store any violation of this constraint, the planners cannot enter this schedule in the system.

  If the system cannot register a violation of a subject domain norm, it makes no sense to require it to enforce compliance, because it cannot remember any violations. Hence, when the 3rd column contains No, the 4th column is empty.

- When an observed violation must be registered by the system, the system may be required to do something about the violation. In many cases, it makes no sense to require the system to do anything about it, because the system has no capability to change the state of affairs. For example, the system cannot do anything about the extra bus parked at a platform, which causes a violation of constraint 1. Just for the sake of the example, I added two examples where the system is required to take action (even though it is questionable whether the system can take a sensible action in these cases). When there is a platform for which no preference is registered (property 9), it can for example inform the database administrator or even add an arbitrary preference. In the case of a violation of property 10, a similar action can be taken.

If the system is required to take action when a constraint is violated, then this should be reflected in the event list. For example, violations of properties 9 and 10 are condition events that should lead to system actions. Because these event–action pairs are rather tenouous, they have not been included in the event list.

To summarize, all of the entries in column 2 with Yes and the entries in column 3 with No are integrity constraints for the data in the system. The entries in column 4 with yes imply some response action by the system, that must be triggered when a condition on the data becomes true. This should be reflected in the event list.

3.9 Assumptions Made By the Requirements

The system as specified will not function as intended if the following assumptions are violated.

1. Bus trips behave in the ideal manner shown in figure A.2.
2. Each entry sensor can sense which particular bus passes it.
3. An entry sensor cannot sense which trip a bus is making.
4. Drivers always follow the instructions of the system. For example, they never park at another platform than the one they are allocated to.

If any of these assumptions is violated, a system that implements the requirements specified here, cannot be guaranteed to work as desired by the stakeholders.

Some of these assumptions are violated very easily by the environment of the system. The first assumption is for example too restrictive for a system that relies upon it to be useful. A bus may pass a sensor that is brought from or returned to its garage, or that is towed by another bus, or that is pushed by its passengers to its destination, or that is hijacked, or that is otherwise occupied in a way that differs from the ideal trip behavior and that makes it meaningless for the system to direct it to a platform of the bus station. Only busses in working order traveling for a trip should be directed to platforms.

On the other hand, we cannot require a sensor to be able to sense the rip that a bus is making, so the system must indeed perform some event recognition activity to figure out which trip a bus is taking. When a bus passes a sensor, the system should use it database to check whether this bus currently is out on a particular trip or not. In the first case, the system may make the assumption that the bus is in normal working order and can be directed to the platform allocated to the trip.

Chapter 4

Essential Architecture

An essential system decomposition is a decomposition defined in terms of the external environment of the system. Decomposition criteria may refer to the subject domain, the external interaction context, external functions, etc. The essential decomposition should remain invariant under changes of implementation.

4.1 Classification and Multiplicity

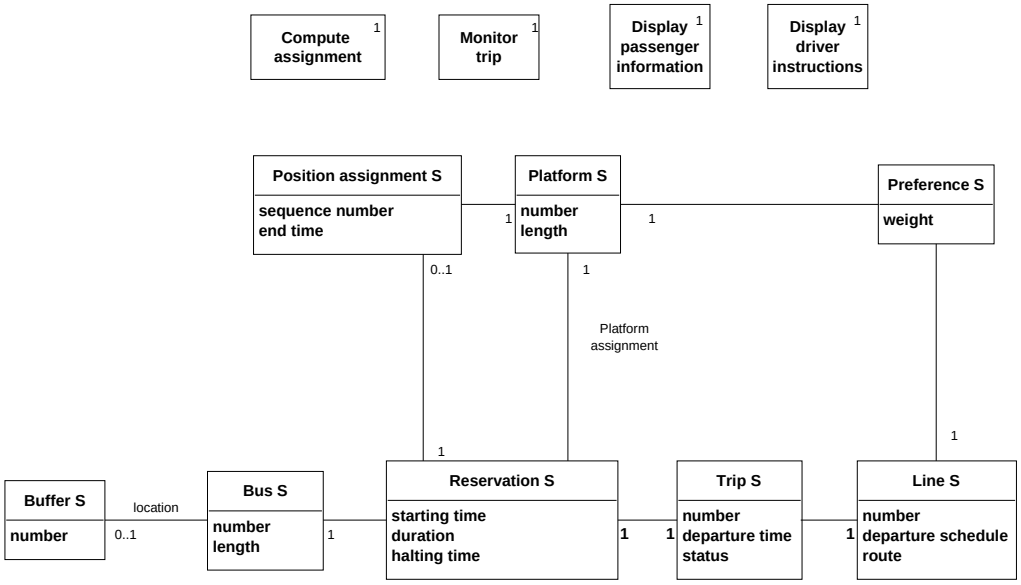

Figure 4.1: Class diagram of essential system decomposition.

Figure 4.1 shows an essential decomposition of the system which duplicates the subject domain decomposition and adds components that handles the desired system functions. The decomposition

is represented by a class diagram. Lines are included to be able to represent multiplicity properties; they do not represent communication links.

Given the simplicity of the system, this diagram is (almost) superfluous to draw. The functional components are taken straight from the function refinement tree. The software objects that represent subject domain objects are called **surrogates**. In order to avoid confusion with their subject domain objects, they have their name suffixed with an -S. The multiplicity of singleton classes (the functions) has been added.

4.2 Communication

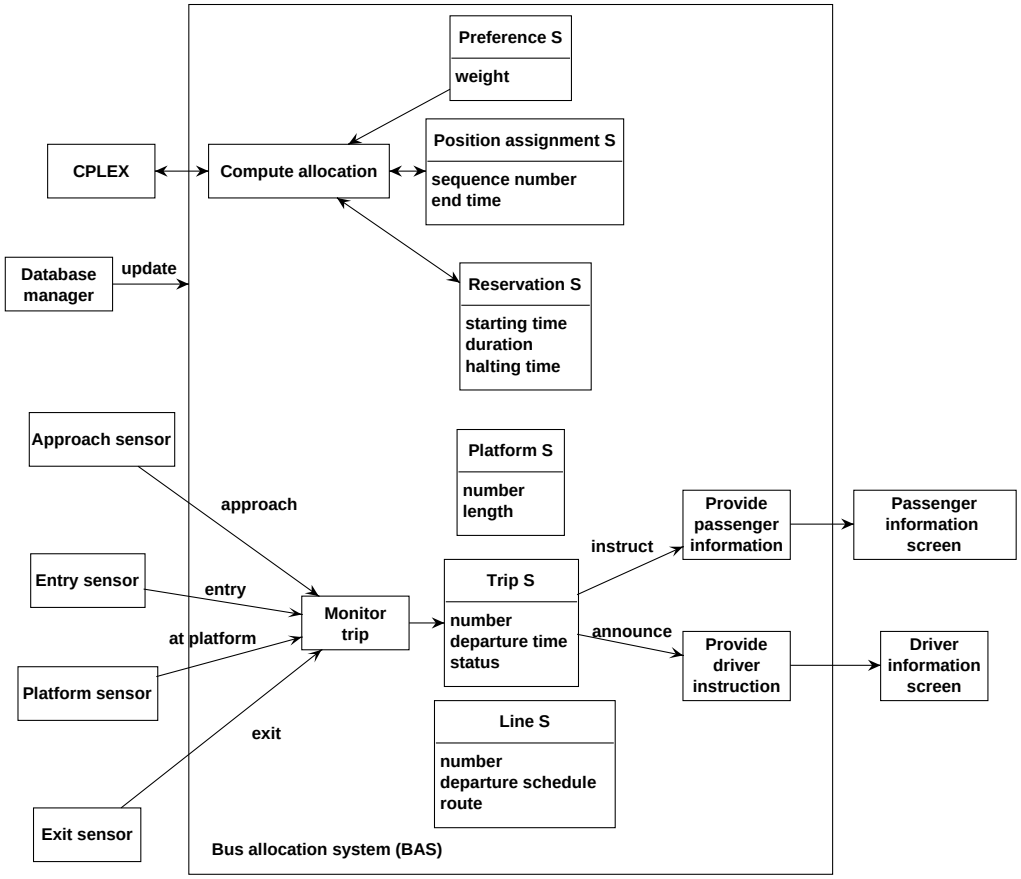

Figure 4.2: Communication diagram of essential architecture.

Figure 4.2 show internal and external signals passed between components of the system and its external environment. It is a refinement and decomposition of the context communication diagram (figure 3.3), that shows the *external* communications: Communications have been specified in a little bit more detail, and the internal components of the system have been added. Arrows represent the direction of the signals. A signal is a message sent from one object to one or more other objects.

Events and actions in a Mealy machine are signals. A read action of one object by another is *not* a signal. Addition of arrows that represent read actions would clutter up the diagram. If necessary, read actions can be represented in a separate diagram.

Chapter 5

Discussion

For lack of information, the implementation architecture of the system has not been shown. It should consist of a deployment diagram, which is a communication diagram showing the network topology of devices and computational resources, and of a traceability table showing the allocation of essential components to network resources.

It is an interesting exercise to replace the object-oriented communication diagram of figure 4.2 by a data-oriented diagram, that distinguishes data storage from data processing. Figure 5.1 is

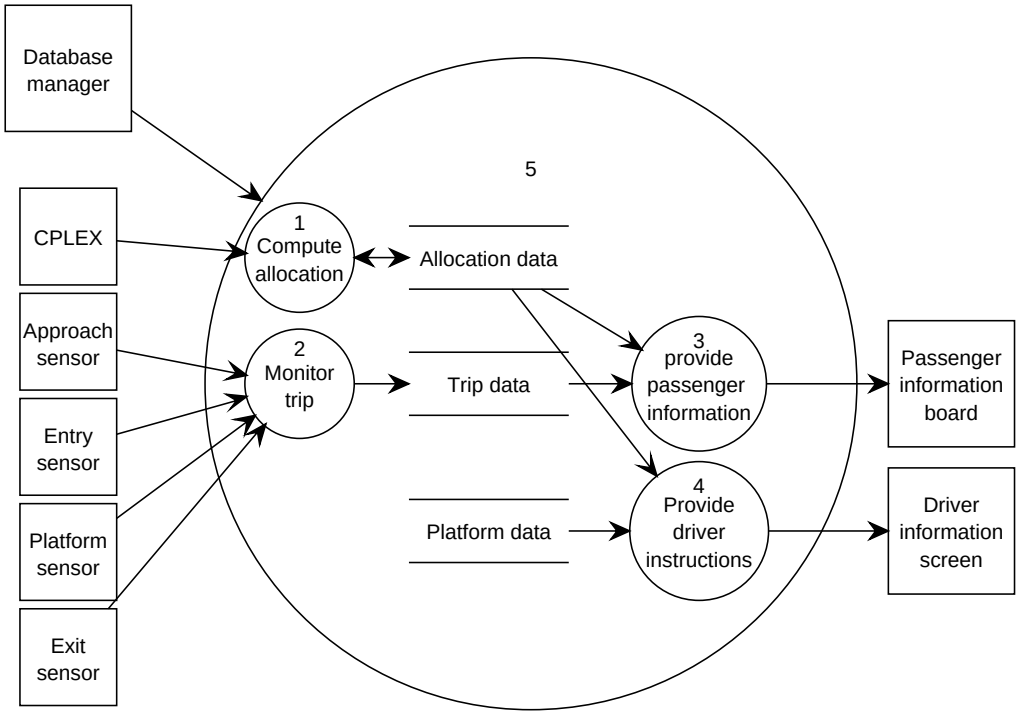

Figure 5.1: Data-oriented communication diagram of essential decomposition. Compare this to figure 4.2.

a DFD that shows a data-oriented decomposition. This is closer to the architecture of a system that would work on top of a database. Where the class diagram in figure 4.1 contains a model of the data in the database as well as of the functional components defined on top of this database, the DFD of figure 5.1 makes a clear separation between functional components (processes) and databases.

Appendix A

Specification Dictionary

A.1 Subject Domain Entities

Figure A.1 shows the classes of entities in the subject domain of the system, and their multiplicities. The external interactions of the system are about these entities.

A.2 Subject Domain Events

Figure A.2 represents the ideal behavior of a trip. The attribute `status` of a trip represents the state in this diagram. It is stated in terms of events that can be observed by the system by means of its sensors.

The model represents ideal, not actual behavior, because it is very well possible that a bus that starts a trip does not take a path through this diagram: It may get stuck, it may drive circles and loop over one particular sensor any number of times, the driver can ignore the instructions, etc. The model therefore describes the behavior of a trip as it should ideally take place. The model is nevertheless useful, because it defines the meaning of the states of a trip.

A.3 Definitions

- The **weight** of a preference is a natural number that represents the preference for (any position at) a platform. The number 0 indicates highest preference, higher natural numbers indicate increasingly lower preferences.
- For each reservation, the **starting time interval** is the interval

  [starting time, starting time + 2 minutes].
- For each reservation, the **departure interval** is the interval

  [starting time + halting time, starting time + halting time + 2 minutes].

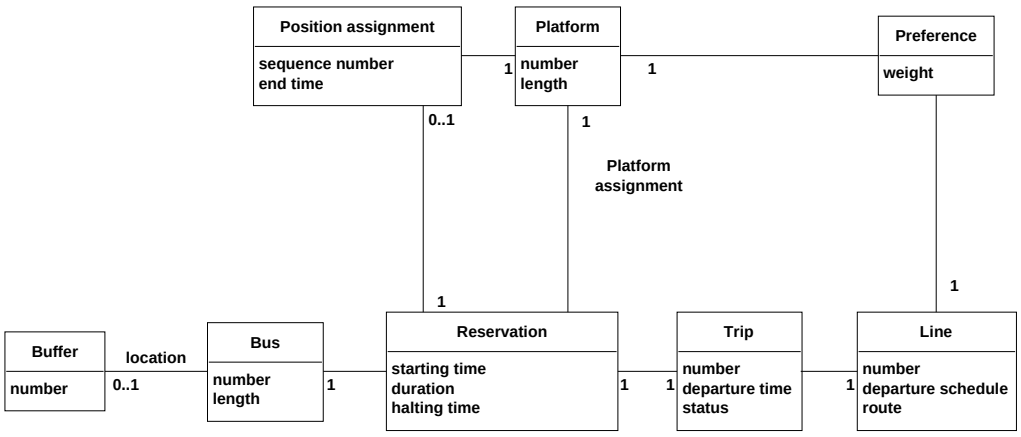

Figure A.1: Class diagram of the subject domain.

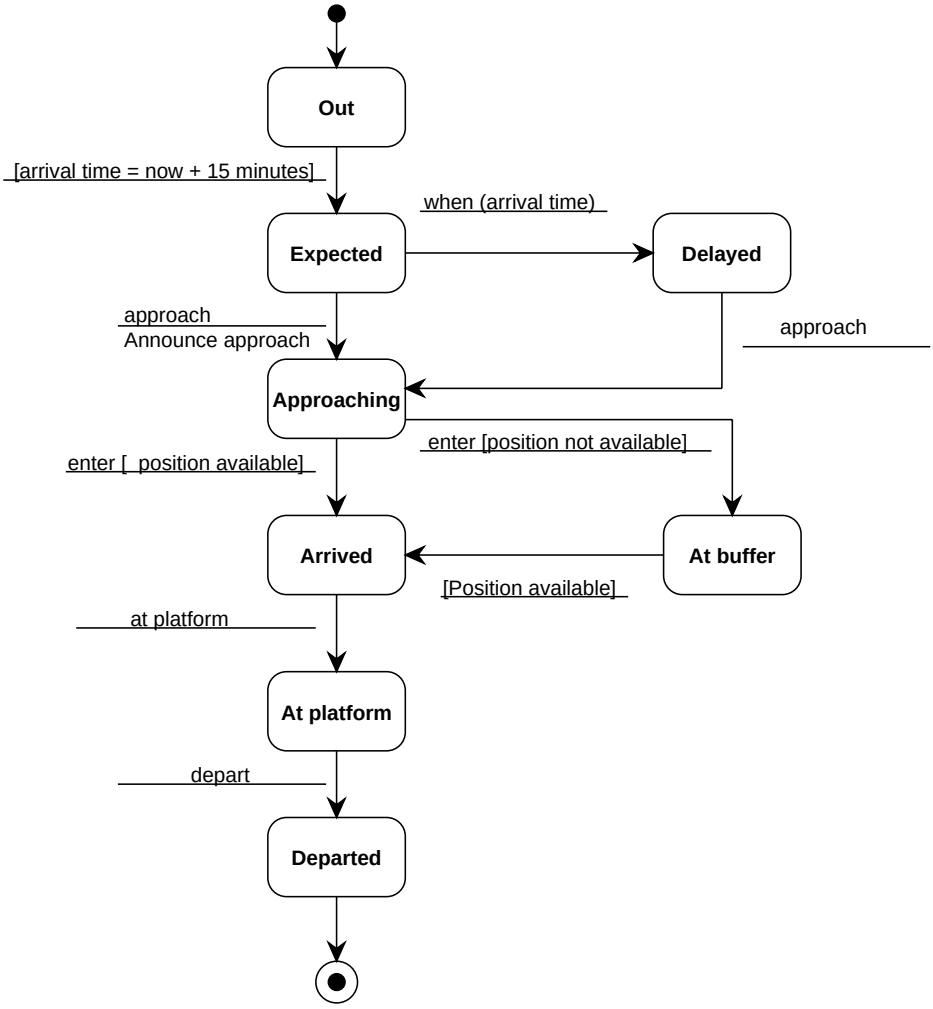

Figure A.2: Mealy diagram of the status of a trip.

Bibliography

[1] A.S. Klusener, S.F.M. van Vlijmen, and A. Schrijver. Compact dynamisch busstation. Technical Report CS-N9601, Centrum for Wiskunde en Informatica, May 1996.

[2] B. van Vlijmen and R.J. Wieringa. Using the Tools in TRADE, I: A Decision Support System for Traffic Light maintenance. Technical Report IR-435, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, November 1997. ftp://ftp.cs.utwente.nl/pub/doc/MAICS/97-TRADE01.ps.Z.

[3] R.J. Wieringa. Using the tools in TRADE, II: Specification and design of a meeting scheduler system. Technical Report IR-436, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, November 1997. ftp://ftp.cs.utwente.nl/pub/doc/MAICS/97-TRADE02.ps.Z.

[4] R.J. Wieringa. Postmodern software design with NYAM: Not yet another method. In M. Broy and B. Rumpe, editors, *Requirements Targeting Software and Systems Engineering*, pages 69–94. Springer, 1998. Lecture Notes in Computer Science 1526.

[5] R.J. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4):459–527, December 1998.