Controlling Automated Traffic Agents

TR 4000
Controlling Automated Traffic Agents

Proefschrift
ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft
op gezag van de Rector Magnificus prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 4 februari 2003 om 10:30 uur
door David Gabriël LINDEIJER

informatica ingenieur

geboren te Delft.
Preface

The research presented in this thesis introduces TrAcEs and FoRcEs as control-software design methods to model autonomous traffic agents using shared infrastructures. This method is supported by a workbench that provides software components which facilitate rapid development of safe, flexible, agile, and scalable, traffic control systems in a modern programming language. The relevance of TrAcEs and FoRcEs is demonstrated via examples, one of which is an envisioned high-performance automated transport system on a container sea-terminal. The required traffic control system is a challenge to design and implement, and is validated within a laboratory for logistic research.

This research is inspired by Joseph J.M. Evers. His passion and vision regarding the control of automated traffic agents, and his drive to realize a fleet of keyboard-sized automatically guided vehicles, was invaluable. It provided an environment in which not only the TrAcEs and FoRcEs control-software design methods could be conceived, but the reference implementation of their software workbenches produced. As a result, this research has not only produced this thesis, but also software that can be deployed to develop real traffic control systems.

I would like to thank Herman Wierenga for his work during development of the envisioned traffic control system, and during development of the mini-agv in particular. Without his enthusiasm the laboratory for logistic research would not have been as good as it is, and with his humor the work was simply fun to do. I am also indebted to Ruud Sommerhalder, who help allot by providing feedback as I was writing this thesis. Not only that, via him I found the opportunity to conduct this research. Furthermore, I would like to thank my colleagues at the section of Transportsystems and Logistics at the Technical University in Delft for providing an amiable environment to work in. Finally, I wish to thank Caroline; I love you, and Isa is beautiful.
Contents

Preface i

1 Introduction 1
   1.1 Aim and Scope of this Study 2
   1.2 Core Questions 4
   1.3 TrAcEs and FoRcEs and MiniWorld 8
   1.4 Contribution of this Study 11
   1.5 Organization of this Thesis 12

2 Background 15
   2.1 Automatic Vehicle Guidance 16
   2.2 Routing and Deadlock 19
   2.3 Large-Scale Automated Transport 22
   2.4 A New Traffic Control Method 28
   2.5 TrAcEs 37
   2.6 FoRcEs 42
   2.7 Conventions 45

3 Agent Communication 47
   3.1 Engineering Software Agents 49
   3.2 Message Exchange 54
   3.3 Event Notification 61
   3.4 Services 65
   3.5 Timing 68
4 Automatic Vehicle Control  71
4.1 The Function Space ................................. 72
4.2 Trace Intelligence ................................. 78
4.3 The Machine Function Driver .................. 88
4.4 Validation ........................................ 97

5 Distributed Traffic Control  99
5.1 Semaphores ...................................... 100
5.2 Distributed Scenes ............................... 106
5.3 Actor ............................................ 117

6 Traffic Intelligence  121
6.1 Advanced Traffic Intelligence .................. 123
6.2 Basic Traffic Control ............................ 138
6.3 Monitoring Action Events ....................... 144
6.4 Exceptions ...................................... 146

7 The Crossover Terminal  149
7.1 The Crossover Layout ........................... 150
7.2 The Crossover Agents ............................ 152
7.3 The Crossover Actions ........................... 156
7.4 Validation in Mini-World ....................... 170

8 Implementation Issues  175
8.1 Implementation of AgileSpace .................. 175
8.2 Distribution of Traffic Control Agents ........ 177
8.3 Distribution of Semaphores .................... 178
8.4 Atomic Reserve .................................. 181
8.5 Nondeterministic Selection ..................... 183
9 Conclusion

9.1 Validation of FoRcEs and TrAcEs ........................................ 191
9.2 Rapid Development of Traffic Control .............................. 192
9.3 Societal Implications .................................................. 194

A Code Excerpts ............................................................ 199

Bibliography ................................................................... 213

Summary .......................................................................... 219

Samenvatting .................................................................... 223

About The Author ............................................................ 227

Glossary ........................................................................... 229
List of Figures

2.1 ECT automatically guided vehicles .................................. 22
2.2 ECT automatic stacking cranes and quay cranes .................. 23
2.3 Layout of the crossover terminal ................................... 25
2.4 The northern side of the crossover area ............................ 26
2.5 The set-up of AgileFrames ......................................... 32
2.6 An intelligent traffic agent ........................................... 34
2.7 A swerving trace for an machine function driver to follow ... 43

3.1 Using Signs and Signals for event notification .................... 63

4.1 The mini-agv, a keyboard size automatically guided vehicle . 71
4.2 A sequence of achieved functions states in XYASpace ............ 73
4.3 Translation and rotation within XYASpace ......................... 76
4.4 Simple traces through XYASpace .................................. 77
4.5 The pilot course ....................................................... 78
4.6 A BrakewayPrecaution ............................................... 82
4.7 Precaution and flags ................................................ 83
4.8 A sequence of Moves ............................................... 90
4.9 Functional decomposition of the MFDriver .......................... 93

5.1 A layout and shared resources ..................................... 102
5.2 Using Tickets to avoid collisions ................................... 104
5.3 A hierarchy of Scenes .............................................. 107
5.4 Request abcdeScene to create a new AtoE .......................... 109
5.5 Method assimilate is invoked upon the downloaded AtoE ..... 110
5.6 Action aToC is downloaded together with aToB and cToC ..... 111
5.7 Action cToE is downloaded together with cToD and dToE ..... 112

6.1 Using Actions & Ticket interlacement .................. 123
6.2 Reserve before insist .................................. 124
6.3 Choice via attempt .................................... 126
6.4 Collecting Tickets .................................... 127
6.5 Stop and go ............................................ 138
6.6 A concurrent point of view .............................. 140

7.1 Layout of the crossover terminal ....................... 149
7.2 FuTraces for for travelling from a stack area to a quay area .. 150
7.3 FuTraces for travelling from a quay area to an agv-turntable . 151
7.4 Guarded Resources ..................................... 153
7.5 The stackAreaToWestToQuayArea[11][1][1] .................. 159
7.6 Infrastructure used by StackAreaToWest Actions .......... 162
7.7 Infrastructure used by StackAreaToEast Actions .......... 164
7.8 Resources used by a StackAreaToWest and a StackAreaToEast .. 164
7.9 Infrastructure used by CrossoverAreaToWest Actions ....... 165
7.10 Required resources for a QuayAreaToQuayCrane .......... 169
Chapter 1

Introduction

Within a global economy, there is an ever increasing demand for handling of materials. There is also an increasing demand to increase reliability and to reduce costs, which can be achieved via automation. To date, there are many successful automated warehouses and factories. However, large-scale automated material handling systems are experiencing performance problems and there are doubts concerning the viability of very large-scale high-performance automated material handling systems.

The performance of a material handling system is formulated with respect to customer requirements: they must be satisfied effectively and efficiently. Performance indicator values for a particular material handling system are determined by its layout and control. In order to improve the performance of a material handling system, its layout and control must be improved. Control can be improved without changing the layout, possibly by introducing new algorithms for planning and scheduling. However, layout can not be improved without changing control, because control must maintain a model of the material handling system it must govern, including a model of the layout. This means that improving the performance of a material handling system implies improving its control, and that better control development methods will facilitate development of high-performance material handling systems.

Three different types of control are distinguished. Logistic control is responsible for cooperation between material handling agents, such as transport agents, storage agents, and transshipment agents. Intelligent logistic control is required to use the material handling system effectively. Traffic control is responsible for synchronizing traffic agents in order to avoid collisions between machines they govern, such as automatically guided vehicles. Intelligent traffic control is required to use a traffic layout efficiently. Machine
control is an agent responsible for reading sensors and setting actuators in order to let a machine perform maneuvers according to the layout. An intricate layout may define maneuvers that are difficult to perform.

Customer requirements change. Therefore, the task of improving the performance of a material handling system, or rather of improving its control, is continuous. A material handling system is agile if its control can be adapted in an expeditions manner at low cost. After customer goods are accepted within the material handling system, the agreed customer requirements must be satisfied. For this reason, control must be flexible in order to adapt to operational circumstances such as congestions or failure of machines. It is essential customer goods are not damaged, and that personnel are not subject to danger; control must be safe. An agile, flexible, and safe, material handling system will be successful, and entrepreneurs will attempt to use its control for larger systems. A material handling system is considered scalable if its control system can deal with any volume of customer goods, infrastructure resources, and traffic agents.

1.1 Aim and Scope of this Study

The aim of this study is to produce a control-software development method that facilitates the implementation of efficient automated traffic control and fine-grained machine control for very large-scale high-performance automated material handling systems. To do this, it must be understood which software components the method should provide, and which current software technologies, such a concurrent programming and distributed programming, can be used to develop such software components. Furthermore, how can current software architectures, such as object-orientation and multi-agents, support the development of agile, scalable, flexible, and safe, automated transport systems? Finally, how can simulation be integrated with development in order to improve the development cycle of automated control of materials handling systems?

The free-ranging automatically guided vehicle (agv) is an exciting machine, and a traffic control-software development method must be suitable for automated transport systems using free-ranging automatically guided vehicles. However, software components provided by the environment must be generic, and vehicle traffic should only be used as a metaphor. Logistic control is considered beyond the scope of this thesis, it is simply assumed that material handling jobs are generated in a manner that is effective with respect to customer requirements.
1.1. AIM AND SCOPE OF THIS STUDY

Context

The design of transport systems for material handling systems is a complex problem, and the design of its overall control system can dramatically affect the system cost and performance [1]. Many interrelated design issues such as vehicle requirements, layout, traffic control, number and location of pick-up and delivery points, vehicle dispatching, vehicle routing, vehicle scheduling, positioning of idle vehicles, battery management, and failure management, must be addressed [2]. However, not all these issues are considered relevant with respect to a traffic control-software development method.

- Vehicle dispatching, vehicle scheduling, positioning of idle vehicles, and battery management, are considered aspects of logistic control that may be considered separate from traffic control and machine control. In general, algorithms and data-structures for efficient and effective planning will not be addressed in this thesis.

- Determining the number and location of pick-up and delivery points is an aspect of determining the layout, which is a creative process beyond the scope of this thesis. However, the freedom provided by a traffic control-software development method to design an arbitrary layout is considered an important issue, because it is strongly coupled to the issue of machine control.

- Vehicle requirements, such as its materials exchange mechanism, physical dimensions, and number of loads it can handle at the same time, is not an issue by itself, but development of control agents for new types of vehicles that can perform difficult maneuvers is considered an important issue.

- Vehicle routing can be found within the contexts of logistic and traffic control. 'Logistic routing' generates routes for transport jobs in order to optimize some value, such as the traveled distance, or the number of required vehicles. In this thesis, the reason for a particular route is not of interest, and logistic routing is beyond the scope of this thesis. 'Traffic routing' allows autonomous selection of alternative routes within the context of a transport job, possibly in order avoid congestions that developed after the transport job was started. Traffic routing is considered important because it is considered essential with respect to flexibility.
• Failure management is an issue that must be addressed in all levels of control. In general, machine and traffic control can only monitor failures, and then trigger some external entity to do something about it.

The department of Transport Engineering and Logistics at the Technical University in Delft also pursues a software development method for logistic-control software systems. Traffic control and machine control are a subsystems within this context. First of all, a general architecture for control software is required. Object-oriented programming suggests borrowing the design for such control-software from the real world. The Council of Logistic Management defines logistic systems as follows.

A network of agents for planning, controlling, and performing efficient and effective flow and storage of goods, services, and related information, for the purpose of conforming to customer requirements.

The control software development method is called AgileFrames, and embraces the notion that a software control system must be a network of software agents. It is assumed that the following three types of agents can be found in any type of automated transport system.

**Machine-Control Agent:** responsible for controlling a machine. An agv-control agent is responsible for controlling the vehicle by reading sensors and setting actuators.

**Traffic Agent:** responsible for guiding the machine-control agent through the infrastructure within the context of a job. An agv-traffic agent is equipped with traffic intelligence about the layout in order to travel to a destination.

**Traffic-Control Agent:** responsible for avoiding collisions between traffic agents if these can not do so themselves, and for providing traffic agents with information about the infrastructure, if so desired.

### 1.2 Core Questions

If a traffic control-software development method is to facilitate the development of free-ranging automated transport systems for material handling systems, then it must be clear what questions the designers of such systems
have. Several issues confronting such designers are presented above. Regarding these issues, questions with respect to agility, flexibility, safety, and scalability, are presented below.

Safety

The transport system of a material handling system is considered safe if automatically guided vehicles do not collide, and if machines do not injure personnel, even if personnel is negligent.

- How can agv-control agents use sensors to maintain distance to other automatically guided vehicles?

- How can concurrent agv-traffic agents be synchronized in order to avoid collisions? Furthermore, how can agv-traffic agents be synchronized without impairing performance.

- How can sensors be used by an agv-control agent to detect personnel and to avoid injuring them.

- Machines, computers, and communication networks fail. How to implement fail-safe behavior for agv-control agents?

Flexibility

The transport system of a material handling system is considered flexible if the use of its layout can be adapted according to operational circumstances. Consider the following operational circumstances.

Traffic congestions: may cause failure to meet agreed customer requirements, such as delivery time. In order to avoid congestions, logistic control must generate different jobs, or the same jobs at a different time. Logistic control is beyond the scope of this thesis, but even if logistic control attempts to avoid congestions, they will still occur. Such spontaneous congestions should be avoided, possibly by allowing an automatically guided vehicle to travel around a spontaneous congestion. To anticipate upon such situations, the layout should define alternative routes that may be used in order to deal with spontaneous congestions.

Handling machine failure: assuming total failure, a failed automatically guided vehicle causes a spontaneous congestion. The traffic control
system must discover such a failure and trigger salvage and repair of the failed automatically guided vehicle. Failure of an automatically guided vehicle may also be partial or eminent, and the traffic control system may be able to avoid failure to meet agreed customer requirements by maneuvering the automatically guided vehicle to a location where it will not cause a spontaneous congestions, and where its salvage is easy.

**Computer network failure:** assuming the failure is partial, then transport activities must continue as long as it is possible and safe to do so.

Traffic agents use the layout according to traffic logic. Flexibility is therefore modelled within the logic traffic agents use, and depends upon the the expressive power of traffic logic. This means the logic should be expressed in a modern programming language.

- How can traffic agents do the following? Anticipate upon traffic conditions, select alternative routes in order to avoid traffic congestions, interact with other agents, such as logistic routing agents, spawn concurrent activities, such as compute a new plan.

- How can traffic agents avoid deadlocks?

- How can traffic agents monitor exceptions, and how can exceptions be dealt with? If an exception is fatal, how can a failed traffic agent be removed from the traffic control system, at run-time, with minimal impact upon ongoing transport activities?

- How can traffic logic be used in a localized manner, in order to allow continued operations within the context of partial failure?

**Agility**

The transport system of a material handling system is considered agile if its control can be adapted in an expeditions manner at low cost, in order to satisfy changing customer requirements. Consider the following three possible changes.

**More customer goods:** surplus system capacity may be able to handle the initial increase of customer goods offered to the system, but eventually a structural solution will be required in order to improve performance:
a new layout and control must be developed. Regarding automated transport, this means that a new traffic layout and corresponding traffic control must be developed.

**New types of customer goods:** the material handling system may need to introduce new machines capable of satisfying new handling requirements. A new type of machine requires a new control system, and this must be developed. Furthermore, traffic control must be adapted to govern the traffic generated by these new types of machines.

**New pick-up and delivery points:** the traffic layout must be redefined to include new infrastructures, and traffic control must be adapted accordingly.

It follows that a transport system is agile if new traffic layouts, together with corresponding machine-control agents, traffic agents, and traffic-control agents, can be developed easily.

- How can traffic-layout design be grafted upon control of automatically guided vehicles, in order to guarantee it only defines maneuvers that agv-control agents can perform?

- Following a virtual lane without collisions does not seem to be a difficult task. Can agv-traffic agents that only intend to follow a virtual lane, be synchronized automatically, without using custom logic provided by developers?

- How can repetition of layout sub-structures be used to define specific traffic control agents components that may be multiply deployed?

- It may not be required to develop the whole traffic control system anew, but only a part. How can traffic control agents be updated, at run-time, with minimal impact upon ongoing transport activities? In particular, how can traffic agents use new traffic control agents without being re-programmed?

- Are there generic components within agv-control agents that are also suitable for other machines?

**Scalability**

A layout sub-divides an area into disjunct infrastructures resources that are shared by vehicles. The transport system of a material handling system
is considered scalable if its design can deal with any number infrastructure resources and vehicles. Consider the two possibilities.

**More infrastructure resources:** a more detailed layout implies more infrastructure resources, or a larger area with similar layout.

**More vehicles:** a more detailed layout may allow more vehicles within a particular area, but in general more vehicles are used to facilitate transport within a larger area.

Scalability can only be achieved if super-linear complexity of traffic control with respect to the number of vehicles and infrastructure resources is avoided. How can this be achieved?

### 1.3 TrAcEs and FoRcEs and MiniWorld

Current control-software development methods for traffic control do not answer, in a satisfactory way, all the questions regarding agility, flexibility, safety, and scalability, asked above. For this reason a new traffic control software development method should be pursued. It is not possible to foresee all possible applications of a traffic control-software development method. For this reason, such a method should focus on being generic and open, in order to facilitate custom solutions for unknown problems. Regarding openness, AgileFrames provides a *communication space* through which arbitrary interaction between software agents may be implemented.

**FoRcEs**

FoRcEs is a software development method, defined within the context of AgileFrames, facilitating the definition of fine-grained control of arbitrary machines. Control of a machine involves three issues. The first issue is specifying what a machine must do. What a machine does is progress through the environment, possibly in a repetitive manner, adding value to customer goods as a side effect. This means such a specification is a sequence of desired states. Because machines operate in continuous-time, it must also be possible to specify a continuous sequence of desired states. The second issue is controlling a machine in order to make it do whatever is specified. The third issue is specifying event-oriented interaction between logistic control and traffic control with the machine's controller. To address these three issues, FoRcEs introduces the following core-notions.
Function Space: a multidimensional space representing the environment a machine functions in.

Trace: a continuous function defining a sequence of desired states through a function space, modelling a maneuver for a machine to perform within the environment. A trace is associated with a velocity profile indicating how fast a machine should progress along the trace.

Precaution: a constraint upon the acceleration with which machine may progresses along a trace.

Flag: an event for monitoring a machine’s progression along a trace.

Move: a traffic control object containing a trace, precautions, and flags. A move can compute the ‘best’ straight line from an arbitrary point in the function space to the trace, and it can compute the ‘best’ acceleration with which the trace must be followed.

Machine Function Driver: a machine’s control agent. It can travel from any point in the function space to another in a straight line. It can ‘perform’ moves by continuously asking the move object to compute the best straight line for it to follow, at the best acceleration.

The FoRcEs development method provides software components, developed in the Java\textsuperscript{TM} programming language, that support the core-notions. The move-component provided by the FoRcEs development method contains a generic algorithm for computing the ‘best’ straight line, it is suitable for implementing the machine function driver of machines for which the notion of following a straight line can be implemented.

In order to facilitate the rapid development of automatically guided vehicle control software, FoRcEs also defines a 3-dimensional function space, called ‘xya’, that represents a flat surface area. An xya-trace may be arbitrary, allowing the definition of an arbitrary traffic layout. An xya-event may be monitored in an asynchronous manner, allowing concurrent programming techniques to be used in order to model safe and efficient utilization of infrastructure resources in an intuitive manner.

TrAcEs

TrAcEs is a software development method, defined within the context of AgileFrames, facilitating the definition of safe, agile, flexible, and scalable,
traffic control systems for traffic within well-structured environments. To achieve this, FoRcEs introduces the following core-notions.

**Semaphore:** a notion borrowed from computer science [3], it 'guards' shared resources and is used for synchronizing traffic agents.

**Ticket:** allows access to shared resources guarded by semaphores to be programmed in a flexible manner.

**Action:** a traffic-control object containing the traffic logic required to travel through an infrastructure in a particular way. An activated action will generate a sequence of moves for a machine function driver to perform. Also, an activated action will be synchronized with respect to other activated actions via semaphores, in this way moves are only generated if required resources are indeed available.

**Scene:** a traffic-control agent governs traffic within a particular infrastructure. A scene agent maintains an ordered set of semaphores, usually guarding shared infrastructure resources, and an ordered set of actions. Scene agents are organized within a hierarchy, facilitating scalable traffic control systems.

**Actor:** a traffic agent that must govern a machine function driver. An actor agent activates actions it has acquired from scene agents. In this way an actor dynamically acquires the knowledge to travel through an infrastructure, facilitating agile traffic control systems.

The TrAcEs development method also provides software components, developed in Java, that support the core-notions within a distributed context. In particular remote polymorphism is applied to the action. This allows an actor agent, residing within an automatically guided vehicle, to 'download' traffic logic from a remote scene agent residing in the infrastructure.

**Mini-World**

Development of TrAcEs and FoRcEs required a laboratory in which to study automated transport systems, and to validate TrAcEs and FoRcEs. Consequently *mini-world* was developed, a facility that includes ten *mini-agvs*, which are keyboard-size automatically guided vehicles. A FoRcEs machine-control agent is developed for the mini-agv, demonstrating how a machine may be controlled, from 'move' to mechatronics. Two traffic control systems
were developed, both of which generate moves that are intended for 'real' automatically guided vehicles. The mini-agv can perform the moves generated by these traffic control systems. The 'crossover scene' demonstrates how a high-performance traffic control system may be implemented, from jobs to moves, that is safe and flexible. The 'abcede scene' demonstrates how a distributed traffic control system may be implemented that is agile and scalable.

The implementation of the mini-agv FoRcEs control system software, and the implementation of the two traffic control systems software, produced the generic software components provided by the workbenches. These generic software components comprise approximately 65% of the implemented mini-agv's controller-code, and may be used for other traffic control systems as well.

1.4 Contribution of this Study

- TrAcEs and FoRcEs contribute to the field of automation of material handling systems by presenting a software architecture for agile, scalable, flexible, and safe, traffic control-software systems, and by providing software components that are not constrained by proprietary licensing.

- TrAcEs and FoRcEs contribute to the field of automated transport systems by demonstrating how remote polymorphism may be used to develop agile traffic control systems.

- TrAcEs and FoRcEs are sub-methods of AgileFrames, a control-software development method for automated logistic systems. AgileFrames contributes to the field of automated logistic-control software engineering by demonstrating how the development cycle of an AgileFrames control system may be shortened by integrating simulation with engineering.

- TrAcEs contributes to the field of logistics by extending the original semaphore concept to allow complex traffic rules to govern access to shared resources. Furthermore, TrAcEs provides 'tickets' that allow access to shared resources, including selection and collection, to be programmed in a flexible manner.

- TrAcEs contributes to the field of computer science by providing 'atomic reserve' with respect to a distributed set of semaphores, this is a powerful tool for avoiding deadlocks within distributed applications.
• FoRcEs contributes to the field of machine control by demonstrating how a machine's activities can be modelled in a function-oriented manner. Furthermore, FoRcEs demonstrates how a discrete, event-oriented, view of a machine progress can associated with its continuous, function-oriented, model.

1.5 Organization of this Thesis

Chapter 2, Background: presents an overview of current automatic transport systems, research in the field of automatic guidance of vehicles, including sensor-based guidance, and current traffic control technologies. It is discussed why a new traffic control design method is needed, and why particular software technologies were chosen for this research. An overview of TrAcEs and FoRcEs is presented that shows how traffic control is programmed using the control-software design methods.

Chapter 3, Agent Communication: presents the technology required to create software agents, in particular messaging, remote method invocation, (asynchronous) event notification, and advertisement/lookup of services provided by agents. It is also discussed how and when such technologies are to be used. The order-queue is also presented, which is a tool for dealing with deadlocks caused by transmission latencies.

Chapter 4, Automatic Vehicle Control: presents FoRcEs, a machine control software development method, and generic software components for rapid development of machine control-software. In is demonstrated how the xya-space may be used to define a virtual lanes for a traffic layout and moves for automatically guided vehicles. The control-software for the mini-agv is also presented, this demonstrates how the generic components can be used.

Chapter 5, Distributed Traffic control: presents TrAcEs, a traffic control software development method, and generic software components for rapid development of a distributed hierarchical traffic control agents called scenes. It is demonstrated how scene agents within the hierarchy may use semaphores to guard shared (infrastructure) resources, and how traffic intelligence, contained within actions may be downloaded from a scene agent to an actor, which is the TrAcEs traffic agent. It is discussed how an actor's lifetime within a transport system evolves.
Chapter 6, Traffic Intelligence: demonstrates how generic TrAcEs software components may be used for rapid development of a flexible traffic control system, within the distributed hierarchical traffic control-software system. In particular, it is shown how to model anticipation, resource (route) selection, collecting (infrastructure) resources, unstructured interaction with other agents, passing resources between actions, preparation, and concurrent activation.

Chapter 7, The Crossover Terminal: demonstrates the development of a high-performance transport system for a container sea-terminal, using the TrAcEs and FoRcEs traffic control-software design methods. In particular, it demonstrates dynamic routing according to resource selection, resources collection via atomic reserve in order to avoid deadlock, generation of a sequence of moves, intuitive definition of moves, and synchronization of traffic processes via semaphores.

Chapter 8, Implementation Issues: presents a discussion regarding the implementation of software components provided by the TrAcEs and FoRcEs software workbenches. In particular, it is discussed how the AgileFrames communication space is implemented, why agents are distributed as they are, how distributed semaphores are accessed by remote prime tickets, how atomic reserve is implemented, and how non-deterministic selection implemented.

Chapter 9, Conclusion: presents a discussion with respect to the questions presented in the introduction regarding safety, flexibility, agility, and scalability. Further work regarding possible software tools to assist in traffic control development is discussed, and societal implications.
Chapter 2

Background

In congested London of 1880, a subterranean transport system using small unmanned trains was built in order to exchange large volumes of post between several large postal stations, without delay. Control was simple: in terminals situated beneath the postal stations, personnel set track-switches, and the trains simply traveled to their destination, through a tunnel, when the power was turned on. A modern version of this successful system is still used today. Technological innovations have provided powerful microcomputer systems and wireless networks which facilitate a new level of transport automation displaying intelligent behavior. Standardization of computer and network technology has made the technology required for automation of materials handling systems available to a wide range of professionals. Combined with the emergence of a global logistics industry, it is believed transport automation will become widespread.

The first type of automatically guided vehicles would follow a fixed route through a material handling system, along which it would visit pick-up and delivery points. Lateral control would be implemented via the physical infrastructure: possibly a track, but usually an inductive guide-wire embedded in the floor. Such systems are still used today, but their major disadvantage is that changing the layout is costly. To address this problem, alternative systems have been developed that allow an automatically guided vehicle to temporarily follow a short virtual lane, but only to return to the embedded wire as soon as possible, or use reflective lines painted on the floor.

In general, free-ranging transport systems are preferred. Such systems instruct automatically guided vehicles to follow virtual lines according to a layout modelled within a computer. It is easier, and cheaper, to adapt a virtual layout than to adapt a physical infrastructure. Free-ranging sys-
tems require some external form of positioning in order to navigate. Two well-known concepts are a grid or triangulation. A grid may be implemented using inductive wires or magnets embedded within the floor. Triangulation is usually implemented using reflected laser beams. Both positioning methods are complemented with internal sensors, such as odometers and inertia gyros, in order to estimate the position in between grid-lines, or when insufficient reflectors are visible.

Vendors of automated transport systems provide different types of vehicles. Examples of common automatically guided vehicles are unit-load vehicles, multi-load vehicles, towing vehicles, fork-lift vehicles, and ‘very narrow aisle’ fork-lift vehicles. Vendors will develop a custom automatically guided vehicle if so desired, such as for in an amusement park. This demonstrates the need to develop new machine-control systems for new types of automatically guided vehicles.

Automation of material handling systems does not restrict itself to automation of the internal transport system, the storage & retrieval systems is also automated, if possible. Use of the automated fork-lift truck may combine automation of transportation and storage & retrieval. In other cases, the storage & retrieval system is a separate machine that interfaces with the transport system via pick-up and delivery points. If so, then the storage & retrieval system has its own private internal transport system using specialized machines, such as automated ‘very narrow aisle’ fork lift trucks, or automatic stacking cranes. The internal traffic within such a stand-alone storage & retrieval system must also be controlled. This demonstrates the need to provide a traffic control system that is generic, and why this thesis only refers to vehicle traffic as a metaphor for interactions between concurrent machines.

2.1 Automatic Vehicle Guidance

Nature is a semi-structured and uncontrolled environment [4]. There is research to develop autonomous robots for such environments, such as the serjoner developed by NASA, which roamed the surface of mars, or automated household appliances such as lawn mowers or vacuum cleaners. In general, such autonomous robots rely on sensors to construct a model of their environment, and do whatever they must do without considering traffic rules.

Another field of research within semi-structured uncontrolled environments is
that of emergent behavior within a fleet of autonomous traffic agents. In this case, all traffic agents are aware of a set of simple behavioral rules, and are capable of recognizing each other. At times, seemingly complex cooperative behavior emerges, with an apparent goal, even though they do not exchange information regarding their intentions [5]. Emergent behavior is beyond the scope of this thesis because a particular goal is pursued, albeit indirectly via a design method, which is high-performance automated transport for material handling systems.

A possible traffic control strategy for semi-structured environments is proposed by [6]. The environment is subdivided into a grid of disjunct areas, and each square area within the grid is associated with a 'communication channel'. A traffic agent must broadcast the path it intends to follow into the channel of its current area. All traffic agents listen for broadcasts in the channel of their current area and all the channels of neighboring areas. If a traffic agent discovers its path intersects with another path and that a collision is eminent, then it must decide, according to traffic rules, if it must give right of way and compute a new intended path. This approach is distributed and similar to how people navigate through an uncharted environment. Note that there must be a common knowledge of traffic rules, how the environment is sub-divided into disjunct areas, and which communication channel is associated with an area.

Well-Structured Environments

Traffic signs are a first attempt to structure the environment and to control traffic agents: a sign identifies a piece of the environment as being an infrastructure for which a particular constraint is imposed upon the traffic agent’s behavior. In this sense, lines on the road and curb-stones are also traffic signs that structure the environment. Traffic signs indicating the right-of-way are special because they also inform concurrent traffic agents how they must interact with respect to each other on a particular shared infrastructure. Traffic lights are simply signs that periodically change, and update, the information for traffic agents.

In order to improve traffic throughput at a particular intersection, the traffic light control system may be equipped with sensors and traffic intelligence in order to adapt its behavior to the traffic load. Extensive research has been conducted in this area [7, 8, 9, 10, 11, 12]. This field of research is beyond the scope of this thesis, because it is oriented upon passenger cars, which are not subject to traffic control in the same way as automated traffic agents.
However, the idea that the infrastructure's traffic signs, or rather its traffic control agents, are equipped with traffic intelligence is a core-notion in this thesis. The key difference is that traffic intelligence, such as traffic rules, is not private to traffic control agents, but shared with traffic agents using the infrastructure. A simple version of this concept, in which the infrastructure-grid shares traffic rules with traffic agents, is suggested in [6].

Another field of extensive research is sensor-based intelligent transport systems on public roads. In this case, intelligence is not associated with traffic control agents associated with the infrastructure, but exclusively with the traffic agents using the infrastructure. Research in this field is briefly discussed below.

**Cruise Control:** a well known, but rather trivial, form of driver assistance in which the vehicle maintains a constant speed indicated by the driver.

**Longitudinal Control:** the vehicle's control agent is responsible for acceleration and must maintain a safe distance to predecessors using sensors [13].

**Lateral Control:** the vehicle's control agent is responsible for keeping the vehicle on the road, usually in combination with longitudinal control with respect to its predecessor. Experiments show that a truck can follow its predecessor on a highway, in a fail-safe manner, using vision-based sensors only. [14]. If lateral control of the leader is also automated, then the road itself must be recognized, possible via standard reflective lines [13], or via a string of embedded magnets [15].

**Crossing Roads:** a vehicle must cross over intersections whilst recognizing vehicles approaching from the sides and avoiding collisions with them. [16, 17].

Pure sensor-based autonomous control of vehicles on public roads is important because it allows automated and non-automated systems to coexist [18], facilitating gradual introduction of automation on public roads. In this thesis, it is assumed traffic agents are equipped with communication facilities to exchange information with an intelligent infrastructure. For this reason, some may argue that this research is not applicable to automation of public roads. This is not the case, because interacting with an intelligent infrastructure will be of value to any vehicle, possibly for navigation assistance, regardless of the degree of sensor-based automation of all other vehicles on the road. This is discussed further in section 9.3.
2.2 ROUTING AND DEADLOCK

Platooning

Platooning is when a group of vehicles merge into a convoy, and the individual vehicle control agents subject themselves to a common 'platoon agent'. The simplest form of platooning uses a manually controlled leader vehicle, which is then the platoon agent, and following vehicles control agents are responsible for the lateral and longitudinal control of their own vehicle with respect to their predecessor. To do this, they must use sensors to observe their predecessor, and exchange information regarding velocity and acceleration. In [14], it is shown that information exchange is not required to follow a predecessor in a fail-safe behavior, but does improve passenger comfort.

An important aspect of platooning is merging a vehicle with the platoon, splitting the platoon, and lane change [15]. In [18], experiments suggest this can be achieved with inter-vehicle communications only, and conventional infrastructure signs such as lane markers and curb-stones. In [15], platooning is implemented using intelligent infrastructure agents that coordinate merging, splitting, and lane changing. The disadvantage of using intelligent infrastructure agents is the added cost for building the road. The advantage is that infrastructure agents can gather traffic data and provide valuable information to traffic agents, such as the layout of the road, the vehicle's location, and the location of neighboring vehicles. After automation becomes widespread, then intelligent infrastructure agents will be able to add value as traffic control agents that improve overall traffic system performance. This concept is discussed in section 9.3.

2.2 Routing and Deadlock

Routing is the problem of finding a route from origin to destination for a transport job. Some transport systems use indirect addressing [1], which is similar to a bus service that follows a fixed route. Whenever goods must be transported it must wait at a pick-up point until an automatically guided vehicle passes by that will transport it to its delivery point. If the delivery point is not serviced by the pick-up automatically guided vehicle, then the goods must be routed through the transport system, possibly switching automatically guided vehicles several times. Routing of automatically guided vehicles is an issue when the transport system used direct addressing, which is when all automatically guided vehicles can travel to all pick-up and deliver points.

Assuming direct addressing of automatically guided vehicles, and an environ-
ment well-structured by a traffic layout, then only a limited number of routes can be selected for a transport job. Static routing does not take current traffic conditions into account, and is often based upon the shortest path criterion [19]. Dynamic routing does take traffic conditions into account, with the intention to avoid congestions. In [20], a dynamic route planner is presented that computes a fixed shortest route that will be congestion-free with respect to all other fixed routes planned up to that point in time. It performs well, but assumes automatically guided vehicles will not fail, and that traffic agents will not deadlock, a problem that will cause congestions and render many previously computed routes useless. Another dynamic route planner that is distributed and incremental is also presented; a traffic agent re-computes its congestion-free shortest route at every fork in the road. This incremental route planner performs poorly compared to the fixed route planner, but is presumed to be superior within a realistic context that is subject to failures.

Another issue that has great impact on routing is the possibility for an automatically guided vehicle to carry multiple loads [1]. Every load may be picked-up and delivered at different locations. In this case, routing is usually intended to minimize the distance traveled by the transport vehicles.

The subject of efficient routing is considered an aspect of logistic control, and beyond the scope of this thesis. However, routing in order to avoid a spontaneous congestions, which is a characteristic of flexible transport systems, is considered important. Furthermore, it must be possible to define a traffic agent that can compute a route for itself, or that can interact with external routing agents.

### Deadlock

Within material handling systems, two types of deadlock are distinguished: *materials deadlock* caused by the inability of automatically guided vehicles to deliver their goods due to full buffers at delivery points, and *traffic deadlock* when automatically guided vehicles wait upon each other forever [1]. Traffic deadlock is of interest to this thesis, although it is not a primary subject. In [21], four conditions for deadlock among concurrent processes are presented: mutual exclusion, hold while waiting, no preemption, and circular wait. Regarding transport systems, the first three conditions always hold [22], and research regarding deadlock in this field is focused on avoiding the fourth condition: circular wait. There are three standard approaches to dealing with traffic deadlocks [1].
Prevention: the layout and traffic control are defined in such a way that deadlocks can not occur. A trivial solution is to reduce the transport system to a single vehicle. More advanced prevention focuses on layout design strategies [23, 24]. For example, a layout should provide many buffer/parking areas where vehicles may wait and where they do not interfere with other traffic, and if possible, traffic should flow in the same direction. Furthermore, routing that tries to avoid congestions, indirectly also tries to avoid deadlocks.

Avoidance: the traffic situation is analyzed dynamically and when deadlock is eminent, it is bypassed by forcing a traffic agent to wait. To do this, deadlocks must be predicted, and several authors use petri-net models [25, 22], another uses a directed graph [26].

Resolution: Deadlock is allowed to occur, after which it will be resolved upon detection. The transport system may be able to detect deadlock automatically via a time-out. Furthermore, the transport system may also be able to resolve the deadlock automatically. If so, then the deadlock is not really a deadlock. If not, then a human controller must be summoned to resolve the new deadlock.

Due to scarce space and minimal budgets to build infrastructures, layout design can not resolve all deadlock problems, and avoidance is the next line of defense. A major disadvantage of deadlock avoidance using petri-nets is that such models are difficult to use. Developing and maintaining such models would be a threat to agility of automated transport systems. This demonstrates the need for an intuitive tool for preventing deadlocks. Finally, resolution is a sensible approach if deadlocks seldom occur and implementing deadlock prevention or deadlock avoidance is expensive. However, in [27], it is argued that deadlock resolution is impossible without specially reserved buffer/parking areas. This suggests that layout design should be given more thought, and demonstrates the need for better traffic control development methods.

Another possible cause for deadlocks is re-ordering of messages exchanged between traffic agents due to transmission latencies. This problem is inherent to distributed control systems and poses a fundamental threat to deadlock prevention and avoidance. This issue is addressed in section 6.2.
2.3 Large-Scale Automated Transport

European Combined Terminals' delta sea-land (DSL) container sea-terminal, depicted in figures 2.1 and 2.2, is a container handling system within a global network of container sea-terminals. Within Rotterdam it is part of a logistic system which can also service other types of transport modalities such trucks and trains. Automatically guided vehicles transport containers from the container stacks to the quay and back, automatic stacking cranes store and retrieve containers from the stacks. The quay cranes are not automated, partly due to legal restrictions. Operational since June 1993, the ECT-DSL terminal was the first automated container sea-terminal in the world.

The ECT-DSL terminal is a truly impressive technological feat, watching the automatically guided vehicles transport containers to and from the quay is simply great. Unfortunately, the ECT-DSL terminal has a performance problem, and resolving these problems has been a continuous task for the ECT. Some of these problems, as stated in 1998 [28], are discussed below.

Regarding automatic vehicle control: front and rear wheels can not be steered independently, and only two types of maneuvers exist: 90° corner, and the s-curve lane switch. Both maneuvers display extreme sway, and traffic control must avoid collisions when automatically guided vehicles take such corners or switch lanes simultaneously.

Regarding traffic control: routing is static and not incremental. As a
2.3. **LARGE-SCALE AUTOMATED TRANSPORT**

Figure 2.2: ECT automatic stacking cranes and quay cranes

result, an automatically guided vehicle will wait for another automatically guided vehicle blocking its route, even if it could ‘easily’ travel around it. Also, it is not possible to select alternative routes at an intersection in order to avoid interfering with other traffic due to the sway. Furthermore, it is not possible to select alternative routes in order to avoid congestions.

Collisions are avoided using dynamic zones that move along together with the automatically guided vehicle. The zone extends a few meters behind an automatically guided vehicle, and a fixed distance in front of the automatically guided vehicle corresponding with its maximum brakeway. In order to avoid stagnation at corners, a complete corner is included within an automatically guided vehicle’s zone before it may take the corner. For this reason, and due to the sway, automatically guided vehicles taking a corner will block use of all other lanes in the vicinity. On straight lanes, an automatically guided vehicle must maintain a large distance from its predecessor, due to the dynamic zones. This distance seems larger than required for safe operations.

Automatically guided vehicles may deadlock in certain areas of the terminal. Merging with, and splitting from, other traffic at the automatic stacking crane areas usually interferes with other traffic on so called ‘highways’. In order to avoid deadlock on highways, a complete highway is included within an automatically guided vehicle’s zone. As a result, automatically guided vehicles can not travel down the same highway simultaneously.

**Regarding information technology:** traffic control is integrated with logistic control, and all control is centralized upon a single computer. The
system can control 50 automatically guided vehicles, but it seems that system resource boundaries have been reached [29]. Furthermore, the whole container terminal is vulnerable to failure of the single computer. Control is not generic, but unique for every type of machine. As a result, implementation and control-software maintenance is more expensive. Because the ECT-DSL’s traffic control system is unique, all improvements must be implemented by the ECT itself, which is very expensive.

The ECT continuously tries to improve the performance of its automated container sea-terminals. It does so by starting internal initiatives, and by participating in initiatives together with external parties, such as project SMAGIC, discussed below. In 2000, the ECT introduced a new version of its traffic control system, called Dynacore, that improved performance and could resolve more types of deadlocks [ff-summer-2002]. Nevertheless, many problems remain. At this time, the ECT has realized its core-business is not software engineering, and has outsourced its automated control systems. Logistic control is now provided by Navis, and traffic control is provided by Siemens-Dematic, which may also use the traffic control system for other automated transport systems.

**SMAGIC: Smarter Guidance For Increased Capacity**

Project SMAGIC [28] was initiated by the ECT, Frog Navigation Systems, and TRAIL, and coordinated by the CTT. The goal of SMAGIC was to discover innovative solutions for optimal use of traffic infrastructure capacity available to automatically guided vehicles. Intended spin-off was improved performance for the ECT’s automated container sea-terminals, and for this reason the problems at the ECT-DSL terminal were analyzed. Some of these problems are introduced above.

TrAcEs was selected as a promising innovative concept for automated traffic control, and a first pilot implementation of TrAcEs, for the purpose of simulation studies, was started. Other promising innovations were definition of alternative maneuvers, and the use of sensors.

In order to compare the value of a TrAcEs-based traffic control system with the traffic control on the ECT-DSL terminal, a functional equivalent TrAcEs traffic control for the ECT-DSL was developed. The TrAcEs ECT-DSL traffic control was not functionally equivalent to the original, but it was considered representative enough. Next, several alternative TrAcEs traffic control
systems were developed, which used dynamic routing and alternative curve types. It was concluded that a TrAcEs traffic control system, with complex semaphore structures, intelligent semaphore claiming strategies, and multiple curve types, may improve performance by 200% - 250%. Furthermore, if sensors are used to maintain distance, then performance may be improved by 300% - 400%.

The pilot implementation of TrAcEs developed for SMAGIC was developed for the purpose of performing simulations, and it served its purpose well. It demonstrated that TrAcEs traffic control systems are flexible and safe, and improve performance. However, the pilot implementation did not satisfy the conditions of agility and scalability. Furthermore, it did not produce code that was suitable for deployment on the actual ECT-DSL terminal, nor did it define a meaningful interaction between the traffic control system and automatically guided vehicles. The TrAcEs implementation presented in this thesis does address these issues.

The SMAGIC project also demonstrated the importance of maneuverability, which is achieved via fine-grained control of automatically guided vehicles. Furthermore, it demonstrated the potential importance of sensors.
FAMAS: First All Modes All Sizes

The economy of scale has driven the development of ever larger container ships. At the same time, container sea-terminal operators are demanding such ships to be serviced in less time. Today, the largest container ship carries 6600 TEU of containers. However, it is believed that jumbo container ships capable of carrying over 10,000 TEU are feasible. Project FAMAS [30] was a research program for development of a new generation automated container terminal for the global market. Project FAMAS was divided into several sub-projects, one of which was the SMAGIC project discussed above.

Within the context of the FAMAS project, a high-performance container terminal was designed for servicing a jumbo container ship within 24 hours. The 'crossover' traffic layout, depicted in figure 2.3, is truly intricate, and implementing a traffic control for the crossover layout is a true challenge. Consider the northern side of the crossover-area layout, depicted in figure 2.4. An automatically guided vehicle exchanges a container with an automatic stacking crane in a northern parking area, which are grouped in fours. An automatically guided vehicle may depart in eastern, western, or southern, direction, depending upon its destination. In extreme conditions, it is possible that twelve neighboring vehicles depart at the same time, all requiring the same infrastructures. The traffic control system must control the such a traffic flow, without overly constraining performance.

A TrAcEs traffic control for the crossover transport system was implemented using the pilot-version developed within the context of the SMAGIC project. This implementation was used to demonstrate the performance characteristics of a crossover transport system. First results indicate high-performance is indeed feasible. In this thesis, a second implementation of the crossover terminal traffic control is presented. This second implementation is interested in other issues: deployable code, and a true interaction between traffic control and automatically guided vehicles.
2.3. LARGE-SCALE AUTOMATED TRANSPORT

OLS: A Logistic System

The OLS project considers a separate infrastructure to facilitate Schiphol airport, which is near to the Dutch capital Amsterdam, the Aalsmeer flower auction, and other logistics service providers in the vicinity. The reason why the Dutch government considers the OLS is because the area around Schiphol airport is severely congested. The largest estimates for the envisioned logistic system is 400 automatically guided vehicles, transporting 3.5 million tons of goods yearly, between 20 terminals, via 25 km of dedicated transport infrastructures [31]. This envisioned logistic system demonstrates the need for a scalable traffic control concept. Furthermore, the available area for individual terminals will be scarce, requiring intricate traffic layouts in order to achieve high-performance. In order to reduce the risk of using new technologies to build such a system, a 1600 m² test site was constructed with scale models and prototypes for automatically guided vehicles and docking stations. In this test site, it was possible to study, under laboratory conditions, driving behavior of automatically guided vehicles, and materials exchange behavior.

Planning and design of the OLS involves many interrelated research subjects, such as traffic layout, traffic management, number and location of pick-up and delivery point, vehicle requirements, vehicle dispatching, vehicle routing, vehicle scheduling, positioning of idle vehicles, battery management, and failure management [2]. In order to aid planning and design, a simulation group was formed to design infrastructure, assess logistics performance, and to test control technology. In order to be robust and extendible, the simulation group embraced a local control concept, within a hierarchical framework, for planning and control [31]. The group concluded that local control does not necessarily impair the logistics performance of the OLS, provided that the local control structures and information exchange between objects are appropriate [31].

Current agv-traffic control technology was considered reliable, but inefficient in its use of space, and space is scarce within the OLS. The TrAcEs traffic control concept, known from the FAMAS-SMAGIC project, was selected as the traffic control concept of choice, because it had already demonstrated to be capable of controlling traffic upon intricate layouts. Furthermore, TrAcEs was selected because local autonomy and scalability are built in [31]. Using the pilot-version of TrAcEs, as developed within the context of FAMAS-SMAGIC as a reference implementation, a new version of TrAcEs was implemented within Simple++, a simulation development environment with extensive facilities for visual programming of the layout of material handling systems.
Using this second implementation of TrAcEs, the simulation group developed a traffic control system for the automated transport at terminals. Simulations using these models were used to derive statistical information regarding average driving speed at terminals, which is strongly dependent upon the traffic layout defined for a terminal. Due to the many interactions at crossing and junctions defined by the two alternative intricate terminal traffic layouts, it would be difficult to infer average driving speed otherwise. The derived statistical data was used to configure a network model of the OLS system, that was used to simulate aspects of logistic control, such as vehicle dispatching, vehicle routing, vehicle scheduling, vehicle battery management, and failure management [2]. Such logistics simulations would in turn produce statistical data regarding arrival patterns of automatically guided vehicles at terminals, which in turn influences average driving speed. This approach demonstrates that traffic control is indeed a separate type of control from logistic control, but it also demonstrates the two types of control are complementary.

The traffic control developed by the simulation group was also used to control traffic at the test site [31]. To do this, the group developed special vehicle-components within the simulation model, that would delegate driving instructions to real automatically guided vehicles in the test site, instead of simulating them. In this way, real automatically guided vehicle traffic at the test site could be controlled using the traffic control modelled within the simulation environment as a prototype. In this way, 9 to 18 months of work was avoided that would be needed to re-implement the ‘real’ traffic control system [31, page 14].

This control situation at the test site was quite curious, because real machines are controlled by a simulation. Usually, the situation is the other way around; machines are ‘emulated’ within a simulation environment in order to test a real control system. This situation demonstrated an essential flaw regarding the second implementation of TrAcEs; it should not have been embedded within Simple++. It should have been developed as a stand-alone traffic control development library, and Simple++ should have been used to emulate the physical OLS system.

### 2.4 A New Traffic Control Method

Many automated small-scale transport systems are operational with warehouses, or so-called flexible manufacturing systems. There exist many vendors of automatically guided vehicles, automatic vehicle-control systems, and traffic control systems. Regarding vendors of traffic control systems, the fol-
lowing two design methods are presented, which are considered interesting and representative. However, there is no intent to express a preference for these two products.

NDC-Training is a design method that allows a customer to automate an existing, manually controlled, transport system. Standard fork-lift trucks are equipped with automated steering and acceleration, and with the Laserway™ positioning system, after which the regular drivers can ‘train’ the automated fork-lift trucks to drive fixed routes and to store/retrieve pallets. The routes are stored within the NDC-Training traffic control system which can also avoid collisions, presumably according to inferred intersections [32].

Siemens-Dematic provides Q-CANDesigner™ as a visual programming tool for developing a layout for an automated transport system. ‘Users can reconfigure their own AGV system without writing special software code or additional vendor support’. The Q-CANDesigner™ application provides input for Q-CAN™, the actual traffic control application that allows the user to input ‘actions’. When layout and control is complete, ‘the program can graphically emulate system logic and performance, highlighting any layout or control areas that need to be changed’ [33].

Visual programming of a layout is very important with respect to agility, and all vendors of automated transport systems should provide such a tool. The current visual programming tools investigated, such as Siemens-Dematic’s QCANDesigner™, NDC’s Laserway™, and AGV-Products’ STAAR™, provide visual components that are ‘clickable’, similar to jigsaw puzzle pieces, which is very easy to use. However, this ease also bars definition of an intricate layout because the visual components can not be arbitrarily small, and can not be overlapped. The reason why the visual components can not be overlapped, seems to be that they represent infrastructure components with exclusive access.

Simulation Environments

There exist several simulation development environments with extensive facilities for visual programming of the layout of material handling systems, such as Simple++™, Arena™, and AutoMod™. These environments provide software components for development of layouts, and for simulating machines such as automatically guided vehicles and storage equipment. Because the components are visual, it is easy to design a material handling system. Traffic layout components are also equipped with traffic control, and it is easy to develop a simulated transport system that avoids collisions.
However, it is not so easy to re-define the control of simulated automatically guided vehicles, nor is it easy to redefine the traffic control logic associated with the layout components. This would be required to study new traffic control concepts such as TrAcEs, as was the case for the OLS project [31]. Furthermore, the simulation environments provide their own programming language in which to re-define components, and such languages can not compete with full-fledged programming languages such as C++ or Java. This is especially true regarding advanced distributed programming techniques such as remote method invocation. For example, Simple++ only allows simulation agents to access external applications via Microsoft-COM\textsuperscript{TM} technology, after which remote applications may be accessed via distributed programming techniques. This is a very cumbersome approach.

Programming the interaction between a simulation and a 'real' control system is not easy. In 1998, the author implemented a simplified but multi-threaded version of TrAcEs in Delphi\textsuperscript{TM}. Then, automatically guided vehicle simulation agents were developed for TOMAS [34], which is a high-performance event-oriented simulation engine for Delphi. The advantage of TOMAS is that simulation code is developed in full-fledged Delphi, and that a simulation model is compiled into machine code. Next, a simple TrAcEs based traffic control system was developed, modelled, compiled, and started, together with a few simulated automatically guided vehicles, all within the same namespace. This failed, because the TrAcEs threads could not interact with the TOMAS thread. The reason for this seemed to be that the TOMAS thread switches between thread contexts, defined for each automatically guided vehicle, on its own accord, and that the threads used for the TrAcEs traffic control could not deal with this. A second attempt succeeded, when the TrAcEs components were re-defined to use the TOMAS thread. This solution was considered unacceptable, and the project was abandoned.

Simulation environments may introduce other problems as well. Profiling is an important debugging tool for software engineers. A profiler indicates how many objects of a particular type are created, and how much time a program spends using objects of a particular type. The TOMAS simulation engine bars profiling. Again, this seems to be because TOMAS switches thread contexts on its own accord, which is essential for event-scheduled simulations. It is not known whether there exist simulation environments that do not introduce this problem, or provide a custom profiler for their particular environment.

It may be that future simulation environments will evolve to equal full-fledged programming environments. However, it seems more likely that
programming environments will extend to incorporate emulation of material handling systems, in a way similar to visual programming of graphical user interfaces. Until that time, 'real' traffic control must be developed within current programming environments, and visualization must be custom-implemented. Fortunately, full-fledged programming languages provide visualization libraries, including 3D-visualization libraries for OpenGL\textsuperscript{TM} and DirectX\textsuperscript{TM}.

Why A New Traffic Control-Software Development Method?

Traffic control development methods provided by vendors of automatic transport systems demonstrate the importance of visual programming with respect to agility, but the methods themselves are not very flexible, and not at all scalable. Furthermore, the methods themselves are constrained by proprietary licensing. Experience in the field of traffic control engineering at the department of Transport Engineering and Logistics allowed development of the TrAcEs traffic control design method, and research provided the possibility to implement TrAcEs for the purpose of simulation studies. However, the problems introduced by current simulation environments, as discussed above, are the reason why it was decided to create yet another, control-oriented, implementation of TrAcEs in a full-fledged programming language. For this new implementation, issues regarding scalability and agility were considered paramount.

In order to achieve scalability, it was clear that a distributed approach to traffic control had to be adopted. Distributed programming also facilitates agility, because distributed systems must consider dealing with partial failures, an issue not relevant to monolithic systems. Indeed, replacing an agent with a new version seems like a temporary partial failure of the agent to other distributed agents. To do this, the Java\textsuperscript{TM} programming language was considered ideal because it is network-oriented. In particular, it provides remote method invocation, which allows development of object-oriented distributed applications. Also, Java provides remote polymorphism, a technology considered essential for agile transport systems. Furthermore, Sun Microsystems introduced Jini\textsuperscript{TM} in 1999, which is a distributed programming method, that was superior to any other similar technology available at that time. Finally, Sun Microsystems also introduced Java3D\textsuperscript{TM} in 1999, which allows development of 3-dimensional visualizations.

Common versions of remote-polymorphism technology is Java-Applets or Microsoft-ActiveX. Examples of this technology being used in practice is
automatic download and installation of a Macromedia-Flash™ plug-in into an internet web-browser after which Flash-movies can be viewed. In this case, the web-page provided by a web-server initiates an upgrade of the web-browser, without requiring the web-surfer using the web-browser to do anything other than to give permission to do so.

An Architecture For New Logistics

Research and development by the department of logistic technology has produced a software development method called AgileFrames, which is intended to facilitate the current automation revolution within the field of logistics. AgileFrames architecture introduces a network of automated logistic agents and a software workbench supporting this architecture in the Java™ programming language. The general architectural set-up is depicted in figure 2.5. An AgileFrames control system exists within a virtual world spawned by a computer network and interacts with external agents. The most important external agent is the customer, who demands goods to be processed by the logistic system. To do this, machines, such as automatically guided vehicles, must store and transport customer goods whilst utilizing the infrastructure. The monitors are external agents responsible for maintaining the automated logistic agents within the AgileFrames control system. AgileFrames embraces the notion of a control system that is subdivided into three complementary sub-systems: a LoGoS sub-system responsible for long-term logistic control, a TrAcEs sub-system responsible for short-term traffic control, and a FoReEs sub-system responsible for real-time machine control.
2.4. A NEW TRAFFIC CONTROL METHOD

Integrating Simulation With Engineering

Simulation is an essential design and development tool for control systems. Even though software simulations may save costs, the development of software simulations is expensive. The AgileFrames software design method allows further costs to be saved by integrating simulation and engineering. This can be achieved by replacing only the external agents of an AgileFrames control system with simulation agents that mimic their behavior. Such agent simulators test the behavior of 'real' LoGoS, TrAcEs, and FoRcEs agents. During development, the simulators and the internal AgileFrames agents progress from simple to complex. Upon deployment the simulators are replaced by their real counterparts; at this time the simulators will mimic the behavior of their real counterparts quite well. By using the AgileFrames approach one can assume, with greater confidence, that the behavior of the traffic control system displayed during simulations will also be displayed after deployment. In this thesis it is shown how this approach is used during the development of the crossover transport system.

After deployment, the external agent simulators are still useful, for two reasons. Firstly, the simulators may be used to test new versions of internal AgileFrames agents as the automated logistic system requires modifications. Secondly, simulators may be used to estimate the future state of the logistic system. To do this, the simulators must be able to function in accelerated time, possibly event-scheduled. If so, the current operational state of the AgileFrames control system must be copied to another environment where the real external agents are replaced by their simulator counterparts. After some time, the simulation environment will have computed a situation that might occur.

A New Architecture For Traffic Control

A concurrent autonomous traffic agent is a machine controlled by a FoRcEs agent called a machine function driver and navigated through the environment by a TrAcEs traffic agent called an actor. Both agents are intelligent and together they can perform jobs generated a LoGoS agent. TrAcEs embraces the notion of a well-structured controlled environment [4] that may be represented as a virtual infrastructure within the virtual world. A virtual infrastructure is a static, but changeable, model of shared (infrastructure) resources. For example, within an industrial, well-structured environment, infrastructures such as roads, intersections, and parking areas, can be represented by semaphores, which spawn a virtual infrastructure within the virtual world.
Figure 2.6: an intelligent traffic agent

The concept for traffic control proposed in [6] is also applicable with respect to a well-structured environment. The environment is not sub-divided into a grid, but into infrastructure components. Traffic agents, or rather TrAcEs actors, must not be aware of the grid, but of the infrastructure. However, the traffic control approach of transmitting intended paths and predicting collisions, seems inefficient in this context because within a well-structured environment the paths are always the same, and the same calculations will be made over and over again. Furthermore, how can the actor know which traffic control rules apply when a collision is eminent? In general, access to a shared resource is granted on a first-come first-serve basis [35]. But if there are parallel parking areas at the entrance of the infrastructure component, such as a ferry, then other traffic rules may be required, such as ‘smallest-vehicle-first’ or random. It seems best that each infrastructure component is not associated with a communication channel, but with a traffic control agent that guards access to its infrastructure component according to traffic rules: a TrAcEs semaphore. If so, then traffic agents must file a request to use the infrastructure component, and the TrAcEs semaphore will honor the request whenever the traffic situation and traffic rules allow. The TrAcEs actor will simply have to wait until that time. After the TrAcEs actor has finished using the infrastructure component, it must inform the TrAcEs semaphore guarding it that the infrastructure component is available once again for use by another TrAcEs actor.

Regarding the behavior of an autonomous traffic agent, three types of intelligence are distinguished: traffic intelligence, trace intelligence, and dexterity. These notions are discussed with respect to figure 2.6.

Traffic Intelligence: required by a TrAcEs actor to navigate through the environment, which is represented by virtual infrastructure spawned
by semaphores, and to generate moves for the machine function driver to perform. Traffic intelligence is complex, and it seems difficult to find a generic form of traffic intelligence that is also suitable for high-performance transport system with an intricate layout. Indeed, how must a generic actor know how to travel through an intricate layout? The shortest route for an individual automatically guided vehicle is not always best from a performance point of view with respect to the transport system as a whole. Furthermore, how can the actor know which semaphores guard shared infrastructure resources, or even more difficult, which semaphores guard abstract shared resources, such as traffic control processes. Finally, how must an actor know how to use a particular infrastructure component?

It seems best to abandon the notion of generic traffic intelligence, and to accept the notion that a TrAcEs actor must be equipped with custom traffic intelligence for using a particular infrastructure. Such traffic intelligence informs an actor of its route, with which semaphores to interact, and which 'moves' to generate for its machine function driver. A move is a machine control object that contains a trace defining a virtual lane through the environment. This approach seems inflexible, and a threat to agility, but this is not true if the technology of remote polymorphism is used. This is discussed below.

**Trace Intelligence:** required by the FoRcEs machine function driver in order to follow a trace by manipulating the machine's controls and observing its sensors. Trace intelligence requires knowledge of the vehicles position and orientation in the environment; it does not require knowledge of traffic rules or intentions of other vehicles.

**Dexterity:** required by the FoRcEs machine function driver to avoid collisions with obstacles. Dealing with obstacles requires a knowledge regarding the position of obstacles within the environment as observed via its sensors, especially in the direction of the trace it is following.

Dexterity and traffic intelligence are complementary approaches to avoiding collisions. Dexterity is suitable for semi-structured uncontrolled environments, and traffic intelligence is suitable for well-structured controlled environments modelled within the virtual world. In figure 2.6, a lower horizontal dotted line corresponds with a more detailed virtual infrastructure and greater responsibility for the actor to avoid collisions. The actor must continuously instruct the machine function driver of the traces it must follow and where to stop. A higher dotted line corresponds with a machine function
driver equipped with the cognitive abilities to decide what to do according to sensor observations. In an unstructured environment the notion dexterity coincides with that of traffic intelligence.

This new traffic control concept considers the physical infrastructure less important than the intelligent infrastructure. Indeed, for free-ranging transport systems the whole point is to avoid embedding routes within the physical infrastructure, but to model the routes within a traffic control agent residing within the virtual world.

**Sensors Versus A Virtual Infrastructure**

Within a structured environment, such as an industrial environment, it seems unreasonable to equip automatically guided vehicles with all kinds of expensive sensors if collisions may be avoided via a virtual infrastructure visible to the TrAcEs actor. Access to a infrastructure component is exclusive if only one traffic agent may use the component at a time. On the other hand, concurrent access to infrastructure components may be allowed if automatically guided vehicles are equipped with sensors in order to avoid collisions within the component [29]. For example, a straight road may be an infrastructure component that several automatically guided vehicles may use at the same time, maintaining distance using forward oriented sensors. Such sensors are not suitable for avoiding collisions at intersections, where traffic may approach from the sides. It seems best to model the intersection as a shared resource with exclusive access, and to model the straight road as a shared resource allowing concurrent access, up to a certain number of automatically guided vehicles, corresponding with the capacity of the road. Note that exclusive access or concurrent access for an infrastructure resource depends upon the characteristics of the vehicle and vehicle control, and not upon characteristics of the infrastructure component.

**Remote Polymorphism And Agility**

Application of remote polymorphism to automated traffic systems provides flexibility and agility with respect to the functionality of traffic agents. For example, This technology allows a traffic agent to download any object that is a descendant-type of an action; an object which the actor knows how to ‘activate’. Descendant types are downloaded from traffic control agents, and contain the traffic intelligence to travel through a particular infrastructure. The traffic agent only knows how to activate this intelligence, and this is enough to guide the vehicle from origin to destination through the
2.5 TrAcEs

The TrAcEs actor is responsible for navigating the machine’s machine function driver through a well-structured controlled environment. The quay of the crossover terminal is structured by the crossover layout, even though the quay itself is an unstructured flat surface. Regarding a particular layout, one can determine shared infrastructure resources such as intersections and parking areas. The actor must avoid collisions upon shared infrastructure resources. To do this, all actors must share a common model of the infrastructure. TrAcEs borrows the notion of a semaphore from computer science to model the capacity of a shared resource and to guard access to it. In computer science semaphores are used to synchronize concurrent programs in order to prevent them ‘crashing’ into each other when sharing data [3, 36]. The innovative aspect of TrAcEs is that collisions between machines are prevented by synchronizing TrAcEs actors, which are programs distributed across the computer network, via distributed semaphores modelling shared infrastructure resources or processes.

TrAcEs introduces several types of semaphores allowing complex prioritized traffic rules to be associated with shared resources. In general, the ‘first-come, first-serve’ traffic rule enforced by the basic semaphore is sufficient to govern access to most resources. For example, an intersection may be guarded by a basic semaphore with a single unit of capacity if automatically guided vehicles only may cross the intersection in the order they arrive, one at a time. Such a semaphore is created for an intersection called X by the following statement.

\[
\text{sIntersectionX} = \text{new BasicSemaphore}(1);
\]

In order to use a resource, a TrAcEs actor must first acquire a prime ticket from the semaphore, indicating the required resource capacity and its priority. The reason why TrAcEs introduces the prime ticket is discussed below. The basic semaphore does not distinguish between actors according to their priority and a prime ticket may be acquired without specifying a priority.
Regarding intersection X, crossing over is an activity requiring the single unit of capacity; a prime ticket to cross over may be acquired as follows.

```java
int x = sIntersectionX.createPrimeTicket(1);
```

Before a TrAcEs actor may instruct its FoRcEs machine function driver to cross over intersection X it must first acquire the resource capacity from sIntersection. The actor demands the semaphore to give it resource capacity as soon as possible by insisting upon its prime ticket. The semaphore will force the actor to wait until that time. After the automatically guided vehicle has passed beyond the intersection, the resource capacity must be returned by freeing the prime ticket. The basic traffic intelligence required to cross over the intersection without collisions, from the direction of A in the direction of B, is modelled in an 'action script' similar to the following.

```java
x.insist();
// cross over intersection X
x.free();
```

The semaphore will give resource capacity to an actor during a ticket.insist according to the available capacity and the priority. If capacity is available the semaphore will give the claimed capacity immediately; if not the semaphore will block all actors during ticket.insist, forcing them to wait. In this way a semaphore synchronizes the actors intending to utilize the resource it guards. Whenever capacity is returned via a ticket.free the semaphore will give the capacity to the next, possibly waiting actor. If two or more actors are blocked within ticket.insist, then traffic rules determines which actor will be given capacity first.

The semaphore is also suitable for guarding abstract resources such as processes. Suppose motorists must drive on the right side of the road and give way to traffic from the right. If four motorists approach a regular intersection at the same time, then all motorists must give way. The issue is resolved after one of the motorists decides to disregard traffic rules and to continue on its way. Computer programs do not disregard rules and four automated motorists would wait upon each other forever. In order to avoid such a deadlock a triple-capacity basic semaphore could be used to guard the traffic process at the intersection. In this way only three motorists would be allowed to compete for the right of way, guaranteeing one of them can always continue on its way.
TrAcEs introduces the *action* as a traffic control object to contain the traffic intelligence required to navigate through an infrastructure without collisions. Suppose the traffic intelligence to cross over intersection X is contained within an action called `crossOverIntersectionX`. The actor may activate the contained traffic intelligence by invoking `execute` upon the action. The invocation will terminate after the automatically guided vehicle has crossed over the intersection. It seems natural to allow actions to activate sub-actions. Suppose action `aToIntersectionX` models traveling from location A to intersection X, and that action `intersectionXtoB` models traveling from intersection X to location B. A a super-action may model the traffic intelligence to travel from A to B via the intersection as follows:

```plaintext
aToIntersectionX .execute();
crossOverIntersectionX .execute();
intersectionXtoB .execute();
```

In general, an action is considered responsible for organizing access to the resources it requires. For `crossOverIntersectionX` this means that it is responsible for acquiring prime ticket `tx` from semaphore `aIntersectionX`, and it is responsible for insisting and freeing the prime ticket before and after the intersection. Recall that `tx` must be insisted before the intersection and must be freed after the intersection. Because the other actions also must insist and free prime tickets in order to use the resources they require, the prime ticket invocations will be interlaced. Furthermore, the timing of these interlaced invocations will depend upon the velocity of the automatically guided vehicle: in order to avoid velocity variations the actor must insist upon `tx` at the time the automatically guided vehicle must start decelerating in order to stop at the intersection. Fortunately it is possible to develop the simple and intuitive action scripts, such as introduced above, by using concurrent programming techniques.

Within action `crossOverIntersectionX` prime ticket `tx` is insisted and freed. In order to avoid deceleration insist must succeed before the automatically guided vehicle must decelerate. Insist may take some time because the intersection is used by another automatically guided vehicle; if the automatically guided vehicle is in a hurry insist should be invoked at an earlier time. This earlier time will be before `crossOverIntersectionX` is activated. This implies the actor will wish to extract `tx` from `crossOverIntersectionX` before activation. This can be achieved via the following statement.

```plaintext
tx = crossOverIntersectionX.getAccessTicket();
```
CHAPTER 2. BACKGROUND

However, it may not be desirable if the actor is blocked during \texttt{tx.insist} at an awkward time. For this reason prime tickets allow actors to \texttt{reserve} resource capacity without blocking, the actor must verify the capacity is granted before actually using the guarded resource by insisting upon the ticket. The following action script models \textit{advanced traffic behavior} that reduces the chance of deceleration for intersection X by reserving upon \texttt{tx} before activation of departing from A. Note that velocity variations with respect to the automatically guided vehicle governed by this particular script may be reduced, but that the performance of the transport system as a whole may be reduced as well.

\begin{verbatim}
  tIntersectionX = crossOverIntersectionX.getAccessTicket();
  tIntersectionX.reserve();
  //
  aToIntersectionX.execute();
  crossOverIntersectionX.execute();
  intersectionXtoB.execute();
\end{verbatim}

Suppose there exists an alternative route from A to B. The actor may wish to select the route to B according to the availability of intersection X. For this reason prime ticket allow actors to \texttt{attempt} acquiring a resource without blocking: the resource capacity will be acquired immediately or not at all. Such advanced traffic intelligence is modelled by the following action script.

\begin{verbatim}
  tIntersectionX = crossOverIntersectionX.getAccessTicket();
  if (tIntersectionX.attempt()) {
    crossOverIntersectionX.execute();
  } else {
    // capacity for intersection X has not been acquired
    // go some other way
  }
\end{verbatim}

Suppose there is a northern route between A and B, and a southern route. The northern route is guarded by semaphore \texttt{sNorth}, and a southern route is guarded by semaphore \texttt{sSouth}. Traveling via the northern route is modelled within an action called \texttt{aToBviaNorth}, and the southern route within \texttt{aToBviaSouth}. Action \texttt{aToBviaNorth} insists and frees a prime ticket \texttt{tNorth} it creates via \texttt{sNorth}, and \texttt{aToBviaSouth} does the same with respect to \texttt{sSouth}. The actor does not prefer one route over the other; it simply wishes to depart as soon as possible. The route should be selected according to which invocation of insist upon \texttt{tNorth} or \texttt{tSouth} succeeds first. Such non-deterministic
behavior is difficult to emulate in code; therefore, the TrAcEs workbench provides the select ticket, which does emulate such non-deterministic selection behavior. Consider the following action script, first tNorth and tSouth are referenced as accessTickets, then the route is selected using a selection ticket.

```java
tNorth = aToBviaNorth.getAccessTicket();
tSouth = aToBviaSouth.getAccessTicket();
stNorthOrSouth = new SelectTicket(tNorth,tSouth);
stNorthOrSouth.insist();
switch (stNorthOrSouth.getSelectedIndex())
    case 0 : {
        aToBviaNorth.execute(); break;
    }
    case 1 : {
        aToBviaSouth.execute(); break;
    }
```

Suppose there exist normal and extra wide automatically guided vehicles. In order to park at a particular space an extra wide automatically guided vehicle must also use the parking space to the right and to the left. The actor must 'collect' all three parking spaces. Naively, one could presume the actor need only insist upon the three resources before instructing the extra wide automatically guided vehicle to park. Unfortunately another actor may also be insisting upon the same three resources, but in a different order. As a result the two scripts may deadlock and neither extra wide automatically guided vehicle will ever park. TrAcEs provides the collect ticket that reserves a set of tickets in an atomic manner. The following action script uses such behavior in order to park the extra wide automatically guided vehicle without the danger of deadlocking.

```java
tLeftSide = new sParkArea[i-1].createPrimeTicket(1);
tMiddle = new sParkArea[ i ].createPrimeTicket(1);
tRightSide = new sParkArea[i+1].createPrimeTicket(1);
tWide = new CollectTicket(tLeftSide,tMiddle,tRightSide);
tWide.insist();
// agv perform move: park in a wide area.
// wait until the agv has departed again
tWide.free();
```

The actor may wish to nest collect tickets in select tickets, and vice-versa. This requirement is the primary reason for introducing the notion of a ticket.
The notion of a prime ticket is introduced because it is particularly useful within the context of distributed programming.

The set of semaphores modelling the capacity of shared resources spawn a virtual infrastructure, and action scripts model how the infrastructure must be used. If the semaphores and associated action scripts all reside upon the same computer, then the traffic control system is centralized. Regarding scalability, a distributed traffic control system is preferred. Therefore, the virtual infrastructure is maintained within a hierarchy of TrAcEs scene agents, which may be distributed across the computer network. A scene agent also maintains a set of actions that may refer to the actions of sub-scene agents, and at the lowest-level actions must take into account the inertia of a machine, such as an automatically guided vehicle, in order to achieve minimal speed variations. In general the higher-level actions are only concerned with routing decisions. Upon assignment of a transport job an automatically guided vehicle's actor must first contact the scene that can provide the action it required to navigate to the job's destination.

It may occur that an infrastructure occurs more than once within the environment. Examples may be double lane intersections, or the repeated substructures within the crossover transport system. In such a case a TrAcEs scene agent template may be defined for a particular sub-infrastructure. Such a template may be used to create any number of identical scene agents which guard the same type of infrastructure, but at different locations.

### 2.6 FoRcEs

The TrAcEs actor instructs the FoRcEs machine function driver what to do by passing it a move that defines the physical activity the machine must perform, and how the two agents must inter-communicate during the performance. FoRcEs defines a physical activity as a trace through a function space, which represents a desired path through the environment within which the machine operates. The machine function driver is equipped with the trace intelligence required to follow a trace by manipulating the controls of the machine. Regarding communication, the actor must inform the machine function driver of constraints regarding speed and acceleration, and the machine function driver must inform the actor of events that are of interest to traffic control.

The function space of an automatically guided vehicle operating upon a flat surface is 3-dimensional: $x$ and $y$ defines its position and $a$ defines its orienta-
2.6. FORCES

Figure 2.7: a swerving trace for an machine function driver to follow

tion. The position and orientation of an automatically guided vehicle changes as its machine function driver turns the steering wheel and presses the accelerator; its desired path is therefore a trace-function $F(u) \rightarrow <x, y, a>$ prescribing how its position and orientation must progress in order to travel from an origin to a destination. The evolution variable $u$ corresponds with the traveled distance, a velocity profile $V(u) \rightarrow [0, 1)$ prescribes how fast the machine function driver should follow a trace.

The layout of the crossover transport system must be defined in the 3-dimensional function space. This tedious task is greatly simplified by using seven simple traces provided by the FoRcEs workbench. For example, consider the swerving path depicted in figure 2.7; it is constructed using the predefined CircularBendRight, Straight, and CircularBendLeft. The construction code is as follows.

```
03  swerveBegin = new XYASpace(x,y,a);
04  swerveTrace =   new CircularBendRight(10,Math.PI/2);
05  swerveTrace.concatenate( new Straight(10) );
06  swerveTrace.concatenate( new CircularBendLeft( 10,Math.PI/2 ) );
```

A TrAcEs actor will instruct a FoRcEs machine function driver to follow a trace within the context of a move as follows: in line 08 the concatenated sub-traces are associated with the move and in line 11 the machine function driver is requested to start following the `swerveTrace`.

```
07  Move swerveMove = ...
08  swerveMove.trace = swerveTrace;
     ...
11  mfDriver.perform(swerveMove);
```

As a machine function driver follows `swerveTrace` it will use resources it shares with other automatically guided vehicles, such as the intersection depicted
in figure 2.7. The virtual infrastructure, which maintains a representation of shared infrastructure resources, is only ‘visible’ to the TrAcEs actor. This implies the actor must indicate points along the trace where the machine function driver may not progress beyond before further notice. To do this, the actor may associate *brakeway precautions* with points along the trace that the machine function driver will not progress beyond before it is removed by the actor. An actor will only remove a precaution if it is safe to do so. Regarding the *swerveTrace*, the machine function driver may not progress beyond a point 19 meters from the beginning, with a maximum deceleration rate of 2.5, as is defined by the *stopBeforeX* precaution in line 08.

```java
07 swerveMove.trace = ...
08 swerveMove.stopBeforeX =
08     new BrakewayPrecaution(swerveMove.trace, 19, 2.5);
09 ...
10 ...
11 agvMfd.perform(swerveMove);
```

The machine function driver is aware of the speed of the automatically guided vehicle via its sensors; it uses this information in order to decelerate according to a brakeway precaution. The actor is not aware of the speed of the vehicle and cannot know when the machine function driver will start decelerating. In order to avoid velocity variations the actor should remove the brakeway precaution before deceleration is required. To this aim the actor may associate *brakeway flags* with the same points as brakeway precautions. The machine function driver will *raise* a brakeway flag at the same time it must start decelerating for the brakeway precaution. The actor will monitor this event and insist upon resource capacity upon its occurrence. If resource capacity is granted immediately, then precaution can also be removed immediately, and the automatically guided vehicle need not decelerate. In this way velocity variations can be avoided. Regarding the *stopBeforeX* precaution of *swerveTrace*, the *xBefore* flag will indicate when the machine function driver starts to decelerate in order to stop at a point 19 meters from its beginning. At this time the actor will start insisting upon prime ticket *tx* acquired for a semaphore guarding the intersection of *swerveTrace* with the other trace.

```java
09 swerveMove.xBefore = new BrakewayFlag(swerveMove.trace, 19, 2.5);
10 ...
11 agvMfd.perform(swerveMove);
12 swerveMove.xBefore.watch();
13 tx.insist();
14 swerveMove.stopBeforeX.remove();
```
Because the actor is not aware of the speed of the vehicle, it cannot know how long the automatically guided vehicle will use the resource after it has removed a brakeway precaution. To this aim the actor may associate *evolution flags* with points along the trace that indicate where utilization of a shared resource ends. The machine function driver will *raise* an evolution flag whenever it passes by the point. The actor will monitor this event and free the resource capacity upon its occurrence. Regarding the intersection of *swerveTrace* with the other trace, the *xBeyond* flag defined in line 10 will indicate when the automatically guided vehicle has passed beyond the intersection, and prime ticket *tX* is freed immediately after this has occurred in line 15.

```
10   swerveMove.xBeyond = new EvolutionFlag(trace,22,2.5);
11   agvMfd.perform(swerveMove);
12   ...
13   ...
14   ...
15   swerveMove.xBeyond.watch();
16   tX.free();
```

### 2.7 Conventions

It is assumed the reader is familiar with *object-oriented* programming. An understanding of *multi-threaded* programming and of *asynchronous event notification*, is advised. The following conventions are used in this thesis.

- Automatically guided vehicle may be abbreviated as ‘agv’.
- Software code is written in the *Courier New* font. Class and interface names start with a capital letter, instance names start with a lower case letter. The names of actions contain "to", the names of moves contain "2", and the names of traces contain "$".
- In general, the AgileFrames architecture defines interfaces rather than classes. The logistic engineer must define classes implementing AgileFrames interfaces. The AgileFrames software workbench provides classes implementing standard behavior for most AgileFrames interfaces. If a class is intended to be extended by the logistic engineer with custom behavior, then such a class is called an *implementation base* and its name will be constructed by appending *IB* to the name of the interface it implements.
• Within text method invocations may be written without braces and actual parameters. If so, and there exist multiple signatures for the method, then any signature could have been used. If actual parameters are used then that particular signature is considered relevant, especially if there are braces without actual parameters.

• The code presented in this thesis is usually extracted from original code for reasons of clarity; the original code often requires all kinds of declarations that are not of interest to the reader. If so, the omitted code will have been replaced by dots. In relevant cases the text will refer to the appendix where the original code can be found.
Chapter 3

Agent Communication

AgileFrames embraces the notion of a network of automated agents cooperating in order to control a logistic system. A logistic engineer developing such automated agents is interested in modelling the communication between automated agents in order for them to cooperate, and implementing their behavior. Furthermore, the logistic engineer imposes several software requirements upon a network of automated agents: it must be distributed, scalable, interactive, agile, and real-time. Software engineering has developed several software technologies which allow these requirements to be met, in particular object-oriented, distributed, and concurrent programming. Beyond software engineering, a software architecture defines the communication between software components. AgileFrames introduces the Agent as a software component representing an automated logistic agent. The TrAcEs and FoRCeS architectures specify how the MDriver, Actor, and Scene traffic control Agents communicate within a network. A traffic control engineer implements traffic control Agents with the behavior required for a particular envisioned transport system.

AgileFrames can not foresee all possible required communications and must facilitate the implementation of custom solutions. The software workbench, implemented in the Java\textsuperscript{TM} programming language, provides Java itself as the ultimate platform for implementing custom solutions. However, AgileFrames also provides a communication space that is accessible from any location within a distributed environment via which Agents may exchange message-objects called Letters in a send-and-forget manner. Because people solve problems by sending each other verbal messages, asynchronous communication using Letters is an intuitive modelling strategy for logistic engineers. From a software engineering point of view, it is not desirable to implement
solutions utilizing Letters only. The primary reason for this is that most Letters are sent in order to receive a reply. The object-oriented communication paradigm of method invocation does support this notion, and an important result in software engineering in the last decade is the extension of the method invocation communication paradigm to distributed environments. In order to avoid burdening logistic engineers with learning the extra programming skills required to utilize Java's remote method invocation, AgileFrames defines the Method, which is a Letter containing the name of a method implemented by the destination Agent, and actual parameters that must be used to invoke the method upon its arrival. After sending a Method-Letter, a reply Letter will be received containing the result of the invocation. Letters and Methods are the subject of section 3.2.

Within an automated logistic system Agents need to notify each other of the occurrence of events: a listener Agent will request a source Agent to send it an event notification upon the occurrence of the event. AgileFrames provides two different types of event notifications, the Sign and the Signal. The listener Agent will request the source Agent to send back a Sign, which is a Letter without content, if it intends to synchronize its internal process with that of the source Agent. The listener Agent will request the source Agent to send back a Signal, which is a Method without a result, if it intends to use a concurrent sub-process to deal with the occurrence of the event. Signs and Signals are the subject of section 3.3.

Within an automated logistic system Agents must advertise their services to customer Agents. To this aim AgileFrames defines the Service which server Agents can advertise, within the AgileFrames communication space. Client Agents can search for a Service conforming to their needs in the communication space. A client may also request AgileSpace for a notification whenever a Service it requires is advertised. Advertisement and lookup of Services is the subject of section 3.4.

Within a concurrent environment, Letter transmission latencies are unpredictable. Within a distributed environment, Letter transmission latencies are longer and even more unpredictable. The problem with transmission latencies is that a listener Agent may receive event notifications in a different order than that these were sent by source Agents. In order to deal with the issue of unintended re-ordering of Letters, AgileFrames introduces the OrderQueue; it is suitable for order maintenance within a concurrent and distributed environment. In section 3.5 it is shown how OrderQueues are used.
3.1 Engineering Software Agents

The logistic engineer imposes several software requirements upon a network of Agents: it must be distributed, scalable, interactive, agile, and real-time. The software engineer will use computer technologies such as object-oriented, distributed, and concurrent programming, to implement Agents displaying the required communications behavior. As is discussed in this section, the intuition of software engineers regarding how such Agents must behave may not coincide with the intuition of logistic engineers.

Distributed Agents

The notion of object-oriented programming is powerful because it is intuitive beyond the realm of software engineering. It is easy for the logistic engineer to perceive a network of logistic agents as a network of Agent-Objects which mimic the behavior of logistic agents. A major achievement in software engineering is the extension of object-oriented programming to distributed programming. This allows software engineers to abandon a message-oriented programming strategy, which corresponds with the way network communications is implemented, and to maintain an object-oriented approach when developing a distributed network of Agents.

The Remote interface represents the notion of distributed object oriented programming within the Java™ programming language. It implies that an Object’s methods may be invoked from any remote location, possibly the other side of the world. It is important for an Agent to possess a unique identity within a distributed environment, and for this reason AgileFrames defines the Agent interface as follows.

```java
interface Agent extends Remote {
    ServiceID getServiceID();
    Service getService(Class serviceClass) ... ;
}
```

ServiceID getServiceID() returns the ServiceID of the Agent. The ServiceID is an Agent’s unique identity within a distributed environment. The notion of a ServiceID is borrowed from the jini distributed programming platform for java.

Service getService(Class serviceClass) returns a Service according to the specified Service-Class.
CHAPTER 3. AGENT COMMUNICATION

The Remote interface also implies that the Object implementing this interface is not 'mobile': it cannot migrate from one computer to another. However, the Remote interface does not imply that the computer hosting the immobile Object is immobile, as is the case with the computer embedded within an automatically guided vehicle which hosts the Actor and MFDriver Agent.

Real-Time And Scalable Communications

The Internet provides a communication infrastructure that transmits messages with great speed, allowing Agents to communicate in 'real time'. Regarding scalability, the Internet is a scalable technology which is easily upgraded to provide the required communication bandwidth between Agents.

Interactive Agents

The requirement that a network of Agents is interactive refers to its communication with people. In general, the technology used to interact with people is the world-wide-web, and the Agents that interact with people are usually web-servers that function as a gateway to a database.

An Agile Network Of Agents

A network of Agents is agile if an Agent can be preplaced with another Agent in an expeditious manner and at low cost. To do this, an Agent must be able to remove the links it maintains with other Agents in the network, and it must be easy for Agents to create new links with other Agents in the network. Fortunately, the Jini™ platform for distributed computing [37] provides advanced techniques for creating and maintaining relationships between software components in a distributed environment subject to partial failure. These techniques are used to create and maintain links between Agents; this issue is discussed in section 3.4, which deals with advertisement of services by Agents to customer Agents.

Implementing Agents

Within the context of distributed programming, socket technology [38] provides an intuitive form of send-and-forget transmission of messages via the Internet. A distributed environment is inherently concurrent, and within the
context of a concurrent environment Agents may synchronize their cooperative activities by waiting for messages sent by other Agents. The receiving Agent is synchronized because it can only pull a message from a socket after the sender Agent has pushed the message into the socket. This approach to Agent synchronization considers an Agent as a sequential process that resembles a person sitting behind a desk processing Letters arriving in the in-box in a sequential manner. As a result, whenever the Agent waits for a Letter it expects to arrive it can not process any other Letters that arrive in the meantime. In general, this is not the intended behavior of an Agent. In particular, this behavior is unacceptable within an automated traffic control system where traffic Agents must react to traffic events which may not occur in a fixed order, as discussed in chapter 6.2. In order to allow an Agent to process Letters in any order one may re-define the Agent’s sequential process to accept Letters in more than one order. In general, such a sequential process will contain many conditional statements and will be cumbersome to maintain. Furthermore, this solution still implies that a Letter can only be processed after processing of the previous Letter is complete. Again, this may be the intended behavior for a particular Agent, but this will not be true in general. In particular, this behavior is unacceptable within an automated traffic control system where traffic Agents must react to traffic events as fast as possible.

In order to allow an Agent to process Letters immediately one must utilize concurrent programming techniques to define concurrent Agent sub-processes that will process a Letter immediately upon arrival, in a concurrent manner with respect to any other sub-processes active within the Agent. Furthermore, the software code is easier to maintain because a concurrent problem is best expressed in a concurrency-oriented language. A standard concurrent programming technique is multi-threading, which allows the definition of light-weight concurrent sub-processes, called Threads. In general, it is not possible to predict the speed with which a Thread executes code with respect to other Threads. This phenomenon is called nondeterminism [39], it may result in unintended Thread interactions and totally unexpected Agent behavior. For the logistic engineer, unintended interactions are a nuisance, because only intended interactions are of interest. It is clear that AgileFrames must alleviate the burden of dealing with unintended Thread interactions between Agents without seriously deteriorating performance, especially not its real-time characteristics. The best example of this is the implementation of Tickets. However, the logistic engineer cannot avoid understanding how multi-threaded programming effects the behavior of an Agent. In particular, the logistic engineer must understand how Threads synchronize in order to
cooperate, because Agents are programs that synchronize to cooperate.

Beyond multi-threading, the logistic engineer must understand the ramifications of utilizing Methods rather than Letters in order to implement communications between distributed Agents. When a receiver Agent receives Letters from concurrent sender Agents, then the receiver Agent must be equipped with enough Threads to process the Letters in a concurrent manner upon arrival. When a server Agent accepts Method invocations from concurrent client Agents then the client Agents must be equipped with enough Threads to invoke the Methods. This is because the client Thread is utilized to invoke the Method upon the server Agent that will compute the result: the client Thread will not be available to the client Agent until it returns with the computed result. During the time a client Thread invokes a server Agent method it is considered a server Thread. This is because a server Agent defines the implementation of its own methods, an essential characteristic of object-oriented programming. As a result a server Agent still has complete control over communication with other Agents even though its methods are invoked by client Threads. Within software engineering asynchronous event notification is considered an important modelling strategy [37, page 560]. Upon occurrence of an event, a source Agent invokes an event handler method upon the listener Agent. AgileFrames provides the Signal for asynchronous event notification: it invokes a method upon the listener Agent. The Signal invokes the method upon the listener Agent with its own Thread and does not return the result. In general, the result of an event handler method is void, a result the source Agent will not be interested in. This implies the source Agent need not maintain a Thread for a Signal. A source Thread simply sends a Signal in a send-and-forget manner as if it were a Letter. As a result, it may be possible to implement a source Agent in a single-threaded manner.

Java

The AgileFrames workbench is grafted upon the Java\textsuperscript{TM} programming language. This language, together with several packages that provide additional functionality, already facilitates all communication requirements. For example, java.lang provides multi-threading and the communication required for thread synchronization, package java.rmi provides remote method invocation, and package net.jini provides a platform for distributed computing. The java programming language is very well suited for development of advanced software systems and is the preferred language for many software engineers. These advanced communication techniques, developed for software engineers, may be considered too complex for use by logistic engineers, and for this rea-
son AgileFrames provides a simplified form of remote method invocation in the form of methods. However, the logistic engineer is free to utilize the advanced communication techniques provided by Java\textsuperscript{TM} and Jini\textsuperscript{TM} and is encouraged to do so.
3.2 Message Exchange

Within a logistic system, a sender agent may wish to send a letter to another destination agent. AgileFrames introduces the letter as a message-object, which can be passed from one agent to another in a send-and-forget manner via the AgileFrames communication space. AgileFrames provides an intelligent communication agent called AgileSpace. Other agents can store and retrieve letters in and from the communication space via AgileSpace. The communication agent is considered intelligent because it hides distribution issues from the logistic engineer and can retrieve letters from the space which match a template. AgileSpace provides the following methods to agents for communication via the AgileFrames communication space.

void AgileSpace.write(Letter letter): writes a letter into the AgileFrames communication space. The letter remains within the space until it is taken from the space via a take operation. Writing a letter into the space is considered to be instantaneous, it does not block the agent.

Letter AgileSpace.readCopy(Letter template): reads a copy of a letter matching the template from the AgileFrames communication space. The copied letter remains within the space. The read operation blocks the agent until a letter matching the template is found within the space.

Letter AgileSpace.read(Letter template): takes a letter matching the template from the AgileFrames communication space. As with reading, the take operation blocks the agent until a letter matching the template is found within the space.

boolean AgileSpace.isLetter(Letter template): tests if a letter matching the template is in the AgileFrames communication space. If so, the invocation returns true, false otherwise. Testing for existence of a letter matching the template within the space is considered to be instantaneous, it does not block the agent.

The notion of a communication space is borrowed from the Linda™[40] programming language, and the notion of exchanging objects instead of messages is borrowed from JavaSpaces™[41], which is an implementation of the Linda communication space in Java™. The AgileFrames communication space is implemented using JavaSpaces™, which also provides remote polymorphism due to its utilization of java-rmi[42]. This implies that a letter may modify the intrinsic behavior of the receiving agent. JavaSpaces™ extends the original notion of matching, as introduced by Linda™ with respect to messages
to extendible objects. In short, an object \( a \) will match another object \( b \) if all public fields of \( a \) match the same fields in \( b \). A field \( a.f \) will match a field \( b.f \) if \( a.f \) is \( \text{null} \) or if \( b.f.equals(a.f) \). AgileFrames borrows its notion of matching from JavaSpaces\textsuperscript{TM}.

**Unique References**

The Letter provided by the AgileFrames workbench requires a sender, a receiver, and the content upon creation. Obviously, in order to send a Letter to another Agent the sender Agent must specify a unique reference to the receiver Agent. AgileSpace can provide such a unique reference to a new Agent via the following method.

\[
\text{ServiceID AgileSpace.createServiceID(String name): returns a new and unique ServiceID for the name. If the name is null then the returned ServiceID is not associated with a name. If another ServiceID was already requested for the name then null is returned.}
\]

After acquiring a unique ServiceID an new Agent must pass the ServiceID to any other Agent intending to send Letters to it. If the new Agent is already aware of the ServiceID of another Agent, then it might do so by sending it a Letter containing the ServiceID. Obviously, Agents must sometimes be created with ServiceIDs referring to other Agents. In general, it is not intuitive for software engineers to refer to other Agents in terms of ServiceIDs. Therefore AgileSpace allows a new Agent to request for the ServiceID provided to another Agent under a particular name.

\[
\text{ServiceID AgileSpace.getServiceID(String name): returns a ServiceID first created for another Agent under the specified name. If AgileSpace does not recognize the name, then the getServiceID operation blocks the Agent until another Agent requested for a new ServiceID under the specified name.}
\]

**Exchanging A Letter**

Suppose two Agents know each other as sender and receiver. In line S3 below the sender Agent creates a Letter for the receiver Agent. In line S4 the sender Agent writes the Letter, containing the content, into the communication space for the receiver Agent to retrieve.
S1 ServiceID sender = ...  
S2 ServiceID receiver = ...  
S3 Letter letter = new Letter(sender, receiver, content);  
S4 AgileSpace.write(letter);

In line D1 below the receiver Agent creates a ‘template Letter’ which describes the Letter to be retrieved from the AgileFrames communication space. The content of the template remains null because the receiver Agent does not know what the content is. Indeed, the receiver Agent wishes to retrieve the Letter for its content. In line D2 the Letter written into the communication space by the sender Agent is retrieved by AgileSpace for the receiver Agent and in line D3 the content is extracted from the Letter.

D1 Letter letterTemplate = new Letter(source, receiver, null);  
D2 Letter letter = AgileSpace.read(letterTemplate);  
D3 Object content = letter.getContent();

During line D2 above AgileSpace will attempt to match the template to all Letters stored within the AgileFrames communication space. First AgileSpace will identify all Letters within the space for which the sender field equals the reference to the sender Agent. Second AgileSpace will identify all Letters found during the first search of which the receiver field equals the reference to the receiver Agent. This second search may produce any number of Letters. If there are several Letters matching the template, then the AgileSpace will retrieve any one of the matching Letters. If there is no Letter matching the template, then the receiver Agent is blocked until a matching Letter is stored in the space.

An Agent may fail; if so the Agent will not retrieve any Letters sent to it. In order to avoid depleting storage resources due to Agent failure, AgileSpace will remove a Letter from the AgileFrames communication space after 24 hours. An origin Agent may request AgileSpace to store a Letter for a longer time, possibly indefinitely. The notion of discarding Letters from the space after a certain amount of time is borrowed from JavaSpaces\textsuperscript{TM}.

The AgileSpace methods write, read, take, and isLetter, are also implemented by the Letter software component. They simply work via AgileSpace and are only intended as a shorthand. AgileFrames also provides several different types of Letters that contain often recurring types of content.

BooleanLetter: the content is a Boolean which may be true or false.

IntegerLetter: the content is an Integer which is natural number.
3.2. MESSAGE EXCHANGE

StringLetter: the content is String which is a text.

In section 6.1 is is shown how a TrAcEs Actor uses a BooleanLetter in order to receive from another Agent a routing decision.

Sending A Letter And Waiting For A Reply

The object-oriented communication paradigm of method invocation is equivalent to two Letter exchanges. The client Agent first sends a Letter to the server Agent specifying which method it wishes the server Agent to invoke upon itself and the actual parameters. Server Agents will take any such Letter sent to it from the communication space, invoke the method upon itself, and write a Letter back to the client Agent containing the result. In general, an Agent will only send a Letter to another Agent if it wants a reply; therefore, method invocation is usually the preferred communication paradigm. AgileFrames introduces the Method as a special type of Letter that implements a simplified form of remote method invocation via the AgileFrames communication space. The client Agent may invoke the Method upon the remote server Agent via the invoke method provided by AgileSpace for this purpose.

Object AgileSpace.invoke(Method method) throws Exception: invokes a method specified within the Method upon the server Agent, using the specified actual parameters. The client Agent is blocked until the result is returned.

Exception: a problem occurred during the execution of the server-side method. Exceptions are discussed below.

AgileSpace can only invoke the method upon the server Agent if it is aware of the server Agent. To this aim a server Agent must register itself with AgileSpace utilizing the following method.

Lease AgileSpace.register(Agent server, ServiceID sID): registers an Agent after which AgileSpace will invoke methods upon it according to Methods sent by client Agents.

The registration returns a Lease which the server Agent must maintain during the time it is prepared to accept method invocations from client Agents. If the Agent allows the lease to expire, then AgileSpace will stop invoking methods upon it according to Methods. The notion of a Lease is borrowed from the jini distributed programming platform for java.
For example, suppose a server Agent implements a method computeSquare, which computes the square value of an Integer value, and it has registered itself with AgileSpace. The client Agent knows the server Agent as server, in line 02 a Method is created specifying the server Agent, the name of the method, and the actual parameter. In line 03 AgileSpace is requested to invoke the remote method and to return the result.

```java
00 ServiceID server = ...
01 Integer x = ...
02 Method computeSquare = new Method(server,"computeSquare",x);
03 Integer square = (Integer)AgileSpace.invoke(computeSquare);
```

It may be true that a client Agent must perform all kinds of other tasks at the same time it intends to invoke a remote method. Because the Method invocation will block until the result returns from the server Agent, which may take a long time, such a client Agent is best programmed utilizing multi-threaded programming techniques. Utilizing the Method within a logistic system has positive and negative aspects. The positive aspect is that the logistic system engineer need not learn distributed programming techniques in order to invoke remote methods, the negative aspect is that regular compile-time checking of Method invocations is not possible.

In section 7.2 it is shown how an Actor navigating within the crossover transport system uses remote method invocation in order to communicate with a LoGoS Agent responsible for logistic control.

**Synchronization**

In natural language *synchronization* usually refers to performing an activity at the same time. Within computer science synchronization refers to exactly the opposite: concurrent programs are synchronized in order to ensure that they do not access computer memory at the same time. Within a network of Agents, it is important to control their synchronization: only then it is possible to implement cooperation between such Agents. Within a TrAcEs traffic control system Actors must be synchronized in order to avoid collisions between automatically guided vehicles. Shares resources could have been modelled as Letters that are stored in the AgileFrames communication space. Whenever an Actor requires use of such a resource it would have to pull the Letter from the space first, and return it to the space after the resource is no longer required. It has been chosen to synchronize Actors via Method invocations upon traffic control components called Semaphores. The reason for
this choice is that a Semaphore, as a software component displaying behavior, is better suited to capture the notion of resource capacity and traffic rules. The Semaphore is discussed in detail in chapter 6.

Exceptions

A method invocation produces a result, possibly void. It may occur that a server-method fails to produce a result upon an invocation, if so it may throw an Exception. The signature of a method must declare which Exceptions the method may throw. A client-method responsible for a server-method invocation must define exception handlers, which deal with any Exception it may catch. For example, suppose the computeSquare method invoked upon server above is declared to throw an Exception whenever the computed result becomes larger than $2^{31} - 1$, which is the largest value which can be represented by an Integer. The client method may exit the java runtime System after it catches such an Exception, or print the value to the screen, as follows.

```
01 try {
02     Integer x = ... 
03     Method computeSquare =
04         new Method(server,"computeSquare",x);
05     Integer square = (Integer)AgileSpace.invoke(computeSquare);
06     // try succeeded, jump over the catch to line 09
07     } catch (Exception) {
08         System.exit(1);
09     }
```

Agent Failure

The server Agent may fail abruptly, without the opportunity to generate Exceptions within the methods it is executing upon request of a client Agent. Assuming such a failure is caused by the failure of the computer platform hosting the server Agent, the server computer platform cannot maintain the network communication channels between itself and the client computer platforms. If so, network communication middleware such as Java's remote method invocation will throw a RemoteException for the client Agent to catch. The client Agent may also fail and the method defined upon the server Agent will never be invoked by the client. In general, a server Agent provides a method as a service to clients and does not consider it important if the client
invokes the method or not. However, a server Agent may impose an order in which its methods may be invoked, and failure of a client Agent may disrupt this order. In this case the server Agent must monitor its own progress and generate an internal Exception if progress is disrupted. Such behavior is beyond the scope of this thesis, but it is simply robust server behavior within a distributed environment subject to partial failure.
3.3 Event Notification

Within a logistic system Agents experience events; it may be true an Agent is interested in events experienced by another Agent. If so the occurrence of the event must somehow be communicated from the source Agent to the listener Agent. The Sign is one of two types of Letters which AgileFrames provides for event notification via the AgileFrames communication space. A listener Agent will create a Sign and pass it to the source Agent via a custom event registration method, and upon occurrence of the event the source Agent will send the Sign back to the event listener as if it were a regular Letter. AgileFrames provides methods for a listener Agent to 'watch' for Signs sent by source Agents.

```java
void AgileSpace.watch(Sign sign) throws EventException: blocks the event listener until the Sign arrives.

EventException: the sign has been aborted by the source Agent. Aborting Signs is discussed below.

int AgileSpace.watch(Sign[] signs) throws EventException: blocks the event listener until any one of the signs is sent back to the event listener via AgileSpace. Upon arrival of a Sign via AgileSpace the method returns the index of the Sign within the parameter list. The index of the first Sign parameter is 0.

EventException: one of the signs has been aborted by the source Agent. Aborting Signs is discussed below.
```

For example, in line 01 of the code below an event listener creates a Sign for monitoring of an event experienced by sourceAgent. In line 02 sourceAgent is requested to send the Sign back upon occurrence of a particular event. In line 03 the listener Agent watches for the Sign and will be blocked until the Sign is sent back by sourceAgent. In line 04 the listener Agent executes code in order to react to the occurrence of the event.

```
01 Sign eventSign = new Sign(sourceAgent,this);
02 eventSource.registerForEvent(eventSign);
03 AgileSpace.watch(eventSign);
04 // react to the occurrence of the event
```

The source Agent may experience a failure; if so it may abort registered Signs. Listener Agents watching an aborted Sign will catch the Exception.
void AgileSpace.abort(Sign sign, Exception exception): aborts the Sign, an EventException containing the sign and the exception is thrown from any watch-invocation for the sign.

If the source Agent fails very abruptly, without the opportunity to abort the Signs listener Agents are watching for, then the lister Agents will watch forever. For this reason AgileSpace also provides watch-methods which operate under a timeout.

void AgileSpace.watch(Sign sign, long timeOutMs) throws EventException:
blocks an event listener until the Sign arrives.

EventException: the sign has been aborted by the source Agent or the specified amount of milliseconds has elapsed. It the timeout occurred, then the message of the contained Exception equals "timeout".

int AgileSpace.watch(Sign[] signs, long timeOutMs) throws EventException:
blocks the event listener until any one of the Signs is sent back to the event listener via AgileSpace. Upon arrival of a Sign via AgileSpace, the method returns the index of the sign within the parameter list. The index of the first Sign parameter is 0.

EventException: the sign has been aborted by the source Agent or the specified amount of milliseconds has elapsed. It the timeout occurred then the message of the contained Exception equals "timeout".

The watch, send, and abort methods are also implemented by the Sign software component. They simply work via AgileSpace and are only intended as a shorthand. These watch and send methods correspond with the methods wait and notify defined upon Java Objects, which are used for defining intended Thread interactions. In the same way the Sign is used to define synchronization of sub-processes within two different Agents, which are modelled using Threads.

AgileFrames introduces the Signal as a second type of Letter which is intended for asynchronous event notification via the AgileFrames communication space. As with Signs, a listener Agent will create a Signal and pass it to the source Agent via a custom event registration method. Upon occurrence of the event the source Agent will send the signal back to the event listener as if it were a regular Method. Unlike Signs, the listener Agent need not watch for a Signal because AgileSpace will invoke an event handler method upon the listener Agent whenever the source Agent sends the Signal.

For example, in the code below the class definition of an asynchronous event lister is presented; it is interested in the same event introduced above. In
3.3. EVENT NOTIFICATION

Figure 3.1: using Signs and Signals for event notification

line 02 the event listener creates a Signal for asynchronous notification of an event experienced by sourceAgent, and in line 03 sourceAgent is requested to send the Signal back upon occurrence of a particular event. Upon occurrence of the event method notify will be invoked.

```java
00  AsynchronousEventListener {
01   public void script() {
02      Signal eventSignal = new Signal(sourceAgent, this, "notify");
03      sourceAgent.registerForEvent(eventSignal);
04   }
05   notify() {
06      // react to the occurrence of the event
07   }
```

Figure 3.1 depicts how the use of Sign and Signals differ. There is no difference for the source Agent, the issue of interest is how the listener Agent must be programmed. If the listener Agent uses a Sign, then it must provide a Thread that will block as it watches for the occurrence of the event. If the listener Agent uses a Signal, then the Thread is provided by AgileSpace. There is no fundamental difference between the two approaches; there are events, and there is code that must be executed upon occurrence of events. The practical difference is how code may be organized, in particular in combination with concurrent programming techniques. Regarding traffic control, the use of Signals allows the 'main' Actor-Thread to request the MFDriver to perform Moves without requiring the same Thread to watch for all events that the MFDriver may generate. In particular, the Actor-Thread does not, nor any other Thread, need to watch for the occurrence of a exceptional event that may never occur. If such an event does occur, the Signal will invoke the code required to deal with the exceptional event.
As with signs, the source agent may fail. If so, it may abort signals passed to it by listener agents.

`AgileSpace.abort(Signal signal, Exception e)` invokes `handle(Exception e)` upon the listener agent, if it implements the `ExceptionListener` interface. This interface is introduced below.

The `send` and `abort` methods are also implemented by the `Signal` software component. They simply work via `AgileSpace` and is only intended as a shorthand. Contrary to a `Sign`, a `Signal` is not used to define synchronization of sub-processes within two different agents. The `Signal` is used to activate a concurrent sub-process within another agent.

The `ExceptionListener` Interface

A listener agent may monitor the occurrence of an event experienced by a source agent. If the source agent fails it may have the opportunity to inform the listener agent of this failure via the following interface.

```java
interface ExceptionListener {
    void handle(Exception e);
}
```

`void handle(Exception e)`: handles the exception.

If the source agent fails very abruptly, without the opportunity to abort the signals, then the `handle` method defined upon listener agents will never be invoked. A listener agent may watch for a `Sign` under a timeout. This notion is not applicable in the same way to `Signals` because there is no invoked `watch` method to throw `Exceptions` from. However, a listener agent may internally monitor the invocation of `Signal` handler methods using multi-threading techniques, and associate a timeout with this monitoring activity.
3.4 Services

AgileFrames introduces the Service as a software component intended to represent a service provided by a server Agent. A Service software component is identified as such by implementing the following interface.

```java
interface Service extends Serializable {
    ServiceID getServiceID();
    ServiceID getServerID();
}
```

Customer Agents must be able to find the Services that other Agents provide. To this aim, Agents must advertise their Services within the AgileFrames communication space and customer Agents must lookup desired Services in the space. To do this, AgileSpace provides the following methods.

```
Lease AgileSpace.register(Service s, Entry[] attr, ServiceID sID): registers a Service after which a customer Agent can lookup the Service, as discussed below. The Agent is allowed to associate a set of Entries with a Service which may distinguish it from the same type of Service provided by other Agents. The notion of an Entry is borrowed from the jini distributed programming platform for Java. The Agent must specify the ServiceID, which is a unique identifier for the Service it provides.

The Service registration returns a Lease, which the Agent must maintain during the time it is prepared to provide the Service. AgileSpace will discard the Service advertisement if the Agent cancels the Lease, or allows the Lease to expire.

Service AgileSpace.lookup(Class sClass, Entry[] attr, ServiceID sID): returns a copy of the specified Service advertised within the AgileFrames communication space that is associated with the specified attributes or serviceID. The returned copy is called a Service contract.
```

When looking up a Service contract, a customer Agent creates a new link between itself and Agent providing the service within the network of logistic Agents. The two Agents will intercommunicate via the Service contract. After completion of the Service, the customer Agent should discard the depleted Service contract.

The MFDriver, Actor, and Scene interfaces are defined as extensions of Service. The TrAcEs and FoRcEs software workbench provide implementation bases,
which are classes that implement these interfaces. The traffic control engineer must extend MFDriverIB, ActorIB, and SceneIB, to advertise their respective MFDriver-Service, Actor-Service, and Scene-Service, within AgileSpace. These Agents can be requested to advertise their Service upon creation.

Protocols

The LoGoS software design method is concerned with protocols which govern the interaction between logistic Agents. The service protocol modelling language may be used to specify the protocol associated with Service contracts. Indeed, a service protocol defined the order in which Agent may exchange method invocations. Software architectures define the interactions between software components; an architecture does not define how a software component must be implemented. Paradoxically, it is the implementation of a software component that determines how it interacts with the other software components in its environment. This paradox is a serious threat to software engineering because it is simply too difficult for software engineers to keep the implementation of software components coherent with respect to the protocols imposed by the architecture; especially during development when protocols are subject to change. The LoGoS service protocol modelling language, and a compiler that produces software components that enforce a particular protocol, are believed to be essential software engineering tools. The logistic engineer will be able to verify the behavior of Agents before these are added to the network. As a result the network of logistic Agents will become more reliable.

Remote References

Java’s remote method invocation introduces the remote reference. A remote reference implements a remote interface that defines a set of methods which a client Agent may invoke upon a server Agent. Regarding service protocols, a remote reference is a Service with a trivial service protocol that does not impose any restrictions upon the order in which a client Agent may invoke remote methods upon a server Agent. A remote reference is also a unique reference to a server Agent. The difference with a ServiceID is that a remote reference is not persistent: every time a server is re-started, possibly due to a failure or maintenance, its remote reference will have changed. In general, server Agents should utilize the same ServiceID every time it is re-started and register its remote reference as a Service under its persistent ServiceID. Assuming a client Agent is aware of the ServiceID of a server Agent it wishes
to invoke remote methods upon, it may acquire a valid remote reference by looking it up via AgileSpace.

Agility

The agility of an automated logistic system is facilitated by an Agent network within which Agents are easily removed and replaced or surpassed by a new generation without needlessly interrupting logistic activities. This can only be achieved if the Service contracts interconnecting Agents are temporary. Upon completion of a Service the contract must be discarded. Whenever the customer Agent requires the Service once again, it must download a new Service contract from AgileSpace. The desired Service may be provided by the same Agent or by any other Agent which can provide the same Service. An old Agent may unregister its Service contracts with AgileSpace, after which customer Agents can no longer lookup its Services: no new Service contracts will be established between the old Agent and new customer Agents. After all running Services provided by the old Agent are completed and the Service contracts discarded, the old Agent may be removed from the system.

Scalability

The Internet has changed our perception of computers. The individual computer is no longer the focus of attention, focus is upon the Services provided by Agents which are accessible via the computer network. Furthermore, the Internet has demonstrated that such a network of computers is scalable, and that the network of Agents it hosts is also scalable. Indeed, an extra Agent need only be created upon a networked computer with adequate resources. Registration of Services is also scalable, as demonstrated by the domain name services within the Internet. Scalability - the number of Agents comprising an automated logistic system, and that of the traffic control system of the logistic handling sub-system in particular - is not compromised by the software architecture of a network of Agents. Scalability of a TrAcEs traffic control system therefore depends upon the architecture of TrAcEs itself.
3.5 Timing

An event experienced by one Agent may be monitored by another Agent. To this aim, the source Agent sends an event-notification to the monitoring Agent in the form of a Letter, possibly a Sign or a Signal. In general, within a concurrent environment one cannot predict the time it will take for a Letter to be transmitted from one Agent to another. Within a distributed environment, which is inherently concurrent, communication latencies are even longer and more unpredictable. Due to communication latencies, an Agent may monitor events in a different order than these were experienced at the source. This ordering problem is inherent because there always will be transmission latencies between the external logistic agents and internal Agents within an AgileFrames control system. One may attempt to correct the order in which Letters arrive at an Agent by introducing a time-stamp indicating the time a Letter was sent. This approach is not preferred because one must delay processing of Letters in order to be sure no tardy Letters arrive. Furthermore, it is not clear how long such a delay must be, nor is it guaranteed the clocks utilized throughout the distributed system are all synchronized within an acceptable margin. Even so, if order is important, then it must be dealt with explicitly.

Maintaining an Order

It is often well known in which order the arrival of event notifications are to be expected according to the arrival of earlier event notifications. For example, automatically guided vehicles will exit from a single lane tunnel in the same order as they entered the tunnel. However, there is no guarantee the event notifications indicating automatically guided vehicles have entered the tunnel were well-ordered; but it is possible to impose physical procedures to guarantee automatically guided vehicles enter a transport system in a particular order. External monitors would be responsible for enforcing such procedures. A traffic control engineer may utilize queues to maintain the order in which event notifications arrive. It is essential to have complete control over pulling and pushing objects from queues; but this is not guaranteed within concurrent environments. AgileFrames introduces the OrderQueue which guarantees objects can be pulled from one queue and then pushed into another in an atomic manner. To do this, two phase transactions must be used, a procedure discussed in section 8.4.
3.5. TIMING

interface OrderQueue {
    void pull(Object object, Transaction txn);
    void push(Object object, Transaction txn);
}

void OrderQueue.pull(Object object, Transaction txn): pulls the object from
the queue whenever it appears at the front of the queue. The Agent will
be blocked until that time. The queue will not allow the next object
to be pulled from the front of the queue until after commit has been
invoked upon the Transaction Object.

void OrderQueue.push(Object object, Transaction txn): pushes the Object into
the end of the queue. The OrderQueue will not allow any other Objects
to be pushed into the OrderQueue until after commit has been invoked
upon the Transaction Object.

In order to transfer an Object from one OrderQueue to another in an atomic
manner, the Agent must first acquire a Transaction via AgileSpace.

Transaction AgileSpace.createTransaction() returns a Transaction. The two
phase transaction will be driven to completion or failure upon invoca-
tion of commit upon the returned Transaction.

The Agent transfers an Object called serviceID from one OrderQueue to another
in an atomic manner as follows

Transaction txn = AgileSpace.createTransaction();
orderQueueA.pull(serviceID,txn);
orderQueueB.push(serviceID,txn);
txn.commit();

In section 6.2 it is shown how transmission latencies may cause a deadlock
at the exit of a tunnel in which automatically guided vehicles can not overt-
take each other. It is also shown how this deadlock may be avoided using
OrderQueues.
Chapter 4

Automatic Vehicle Control

Figure 4.1: the mini-agv, a keyboard size automatically guided vehicle

Automatically guided vehicles travel through the environment as they perform transport jobs. To do this, a path through the environment must be determined and collisions avoided. AgileFrames introduces the FuTrace as a software component representing a path through the environment, it defines a virtual lane. The FoRcEs MFDriver is responsible for following a FuTrace by controlling the automatically guided vehicle within the environment. The TrAcEs Actor is responsible for determining a FuTrace as it navigates through the virtual infrastructure according to its traffic intelligence. How to define a virtual infrastructure is the subject of chapter 5, and how to model traffic intelligence is the subject of chapter 6. In order to avoid collisions the Actor will wish to indicate points along a FuTrace the MFDriver may not progress beyond without further notice. To do this, the
Actor may associate a BrakewayPrecaution with an FuTrace that specifies how the MFDriver must decelerate in order not to progress beyond that particular point, the Actor will remove the Precaution whenever it acquires permission to progress beyond the point. In order to know when the time has come to remove a BrakewayPrecaution the Actor may associate a BrakewayFlag with the FuTrace that must be raised by the MFDriver at the time the automatically guided vehicle must start decelerating in order to stop according to the BrakewayPrecaution. Upon 'seeing' the raised Flag the Actor will start organizing permission to pass by the point. The Actor will also wish to know when the MFDriver no longer requires a particular shared resource, which will be after it has progressed beyond a particular point. To do this, the Actor may associate an EvolutionFlag with a FuTrace that the MFDriver will raise whenever it progresses beyond a particular point.

Avoiding collisions is not the sole responsibility of the Actor, it shares this responsibility with the MFDriver. For example, the MFDriver will be able to maintain its distance to a predecessor within a tunnel using a forward oriented distance sensor. Unfortunately, the MFDriver will not be able to avoid collisions with crossing traffic at an intersection using the same sensor, and the Actor will be responsible in this case. In order for the Actor to do this the traffic control engineer must represent the intersection as a single-user resource within the virtual infrastructure visible to the Actor. How such a virtual infrastructure is defined, and how the Actor interacts with such a virtual infrastructure, is the subject of chapters 5 and 6. In general, the more sensors a MFDriver may use, the more responsibility for avoiding collisions it will have. In the extreme, the MFDriver will be responsible for avoiding all collisions.

FoRcEs has been under development since 1999, and this chapter presents collaborative work with J.J.M. Evers.

4.1 The Function Space

FoRcEs is concerned with modelling a machine's function. The function of an automatically guided vehicle is to follow paths through a flat-surface environment as it transports customer goods. FoRcEs introduces XYASpace as a function space representing a flat-surface environment. The notion of a function space is discussed below. Within XYASpace the functional state of an automatically guided vehicle is a tuple \((x, y, a)\), describing its position and orientation. A FuTrace through XYASpace defines a sequence of \((x, y, a)\) tuples representing a path through the flat surface environment. XYASpace is rather
simple but possesses enough expressive power to capture the functional state of an automatically guided vehicle and to define FuTraces modelling intricate layouts, such as that of the crossover transport system depicted in figure 7.1. However, for another type of machine, such as an airplane, it would be necessary to capture the functional state with more attributes, such as altitude.

Within the context of XYASpace, the automatically guided vehicle progresses from one functional state to another in a continuous manner as the MPDriver manipulates its controls. A sequence of functional states can be represented as a mathematical function $G$ on a real-numbered interval $[0, U]$ to XYASpace, as depicted in figure 4.2. By definition $G(0)$ is the origin of the automatically guided vehicle, $G(U)$ is the destination of the automatically guided vehicle, and $U$ corresponds with the traveled distance. All $G(u)$ with $0 \leq u \leq U$ represent functional states realized by the automatically guided vehicle as it traveled the distance, $u$ is called the evolution variable. Suppose the function $u(t)$ represents the level of evolution achieved by an automatically guided vehicle in time as it traveled the distance. If so, the progression of its functional state in time is represented by $G(u(t))$. As the automatically guided vehicle follows its path through the environment the progression of the functional state $G(u)$ has been sequential, implying that $u(t)$ is a continuous non-decreasing function of time. The left derivative $\partial u(t)/\partial t$ corresponds with the velocity of the automatically guided vehicle as it followed the path, if it exists.

Representing the path an automatically guided vehicle followed through the environment in the past is not particularly interesting. In order to transport customer goods it must be possible to prescribe the path through the environment that must be followed in the future, and how fast such a prescribed should be followed. For an automatically guided vehicle, FoRcEs introduces the FuTrace, which is discussed in more detail below, as a function $F : [0, U]$ \rightarrow
XYASpace that is associated with a velocity profile $V : [0, U] \to [0, 1)$. The velocity profile indicates how fast the FuTrace should be followed.

the function space

XYASpace is a particular type of function space representing a flat-surface environment. In general, a particular type of function space defines a set of functional states for a particular type of machine that are of interest to its user, which is a MFDriver. During its lifetime, a machine will progress from one functional state to another as the MFDriver manipulates its controls. The MFDriver will do so according to a FuTrace, which defines a sequence of intended functional states that correspond with a job. Due to all kinds of physical conditions and inevitable control inaccuracies the MFDriver can never follow the FuTrace exactly and must continuously try to progress in the direction of the FuTrace and to decrease its distance from the FuTrace. This implies the notion of distance between arbitrary functional states must be well-defined. For this reason a function space, which is represented by FuSpace, must conform to the following conditions.

state distance: $D^s : \text{FuSpace} \times \text{FuSpace} \to \mathbb{R}$.

evolution distance: $D^u : \text{FuSpace} \times \text{FuSpace} \to \mathbb{R}$.

triangle inequality: for both $D^s$ and $D^u$.

discrimination inequality: for all $s_1, s_2$ in FuSpace: $D^s(s_1, s_2) \neq 0$ if and only if $s_1 \neq s_2$.

partiality of $D^u$: for all $s_1, s_2$ in FuSpace: $D^u(s_1, s_2) \leq D^s(s_1, s_2)$.

Due to control inaccuracies, the observed functional state of an automatically guided vehicle will usually not be a functional state prescribed by a FuTrace. The notion of a straight path, which is represented by FuPath, is required in order to specify how the machine must progress from a functional state near to the FuTrace to a desired functional state upon the FuTrace. Because 'near' is not a well defined notion, such a straight path must exist between arbitrary functional states, and it must be possible to interpolate any functional state between the two arbitrary functional states. For this reason a function space must also conform to the following condition.

existence of straight paths: for all $s_1, s_2$ in FuSpace, there exists a path $F : [0, U] \to \text{FuSpace}$ for which $F(0) = s_1$, $F(U) = s_2$, and $U =
D^u(s_1, s_2), such that for all u in [0, U] it is true that D^u(s_1, F(u)) = u and
D^s(s_1, F(u)) + D^s(F(u), s_2) = D^s(s_1, s_2).

FoRcEs provides XYASpace in order to facilitate modelling the function of automatically guided vehicles operation upon a flat surface area, it is defined below. Note that XYASpace conforms to the requirements regarding distance imposed upon a FuSpace and there exist a straight path between two arbitrary points.

class XYASpace extends FuSpace {
    double x,y,a;
    double lambda;
    double stateDistance(XYASpace s1,XYASpace s2) { ... }
    double evolutionDistance(XYASpace s1,XYASpace s2) { ... }
}

stateDistance(s1,s2): √((s_1.x - s_2.x)^2 + (s_1.y - s_2.y)^2 + λ*abs((s_1.a - s_2.a))
evolutionistance(s1,s2): returns √((s_1.x - s_2.x)^2 + (s_1.y - s_2.y)^2)

a trace through a function space

A FuTrace is intended to model how the functional state of a machine should progress in time. This means the sequence of functional states specified by a FuTrace must also reflect this notion of progression. In general, a FuTrace must not define something like a scattergram. Therefore, the desired path defined by a FuTrace must conform to the following requirement.

F : [0, U] → FuSpace is a FuTrace iff: for all ζ > 1 there exists a η > 0 such that, for all u_1 < u_2 < u_1 + η in [0, U]: u_2 - u_1 ≤ D^u(F(u_2), F(u_1)) ≤ ζ*(u_2 - u_1)

In order to follow a FuTrace, the MFDriver must look some distance ahead along the FuTrace. If the MFDriver is almost at the end of a FuTrace, it is not clear where to look. To this aim a FuTrace also defines a trace extension, which is simply a straight line from the end point F(U) through another functional state within its FuSpace. For example, in XYASpace the extension is usually defined in the direction of the orientation of the end point. The velocity profile of the extension is 0. The FuTrace software component is defined below. It is the responsibility of a traffic control engineer to ensure that the implementation of a FuTrace conforms to the imposed requirements introduced above.
abstract class FuTrace {
...
  public FuSpace getPoint(double u);
  public FuSpace getEndPoint();
  public double getEndEvolution();
  public double getProfileVelocity(double u);
}

getPoint returns $F(u)$ of the defined FuTrace, $u > 0$. The FuTrace is defined with respect to $[0, U]$, if $u > U$ then the returned point is computed according to the extension.

getEndPoint returns $F(U)$ of the defined FuTrace, which is defined with respect to $[0, U]$

getEndEvolution returns $U$ of the defined FuTrace, which is defined with respect to $[0, U]$

getProfileVelocity returns $V(u)$ of the defined FuTrace, $u > 0$. $V(u)$ maximum velocity is the desired velocity. The maximum velocity is a traffic intelligence parameter set by the Actor during construction of a Move, an issue discussed below.

transformations within a function space

FuTraces in XYASpace are used to define the layout of the crossover transport system depicted in figure 7.1. As one can see, the layout has a repetitive structure and it would be practical if FuTraces defined with respect to one location were re-used at another location. To this aim, FoRceS introduces the FuTransform, which represents the notion of a transform within FuSpace. In order to facilitate transformations in XYASpace, FoRceS provides
the XYATransform, which implements translation and rotation. It is defined below. The use of XYATransform to translate and rotate a CircularBendLeft, which is a predefined FuTrace through XYASpace provided by FoRcEs, is depicted in figure 4.3.

class XYATransform {
    double x, y, a;
    XYASpace transform(XYASpace s) { ... }
}

transform: returns s, after first translating with respect to x and y, and then rotating with respect to a.

predefined FuTraces in XYASpace

In order to facilitate the definition of FuTraces in XYASpace, FoRcEs predefines seven simple FuTraces within XYASpace; these are depicted in figure 4.4. The definition of CircularBendLeft is as follows.

class CircularBendLeft extends FuTrace {
    ...
    XYATransform xyaTransform;
    public CircularBendLeft(double r, double a, XYASpace beginLocale)
    public double getPoint(double u) { ... }
    public double getEndPoint() { ... }
}

CircularBendLeft: constructs a new CircularBendLeft according the the specified radius r and end-angle a. The new FuTrace is defined with respect to \([0, 2\pi r]\). The transform of this FuTrace is created such that \(F(0)\) equals beginLocale.
getPoint: returns xyATransform.transform(new XYASpace(x,y,a)), where \( x = r \times \sin(u/r), y = r - r \times \cos(u/r), a = u/r. \)

Figure 2.7 of chapter 2 depicts a complex FuTrace constructed from three instances of the predefined simple FuTraces provided by FoRcEs. On the same page as the figure the code for constructing the swervec-trace is also presented.

4.2 Trace Intelligence

The Actor is equipped with the traffic intelligence required to navigate through the virtual infrastructure and to determine the FuTrace the MDriver must follow. The MDriver is equipped with the trace intelligence required to follow a FuTrace by manipulating the controls and observing its sensors. Due to all kinds of physical conditions and inevitable control inaccuracies, the observed functional state, inferred from sensor observations at time \( t \), will be a functional state 'near' to the FuTrace, as depicted in figure 4.5 as \( G(t) \). The MDriver must continuously try to progress in the direction of the FuTrace and to decrease its deviation from the FuTrace. To do this, the MDriver must somehow compute a pilot functional state upon the arbitrary FuTrace, as depicted in figure 4.5 as \( F(Up) \). The computed pilot functional state should depend upon the deviation, the type of machine, and the type of FuTrace being followed. This implies the pilot functional state may be an arbitrary distance from the observed functional state. Therefore, it is best to represent the 'solution' as a FuPath called the pilot path.

How the pilot path is computed affects the behavior of the automatically guided vehicle within the environment, implying that the Actor will wish to control how the pilot path is computed. For this reason an Actor encapsu-
lates a FuTrace within a software component called a Move, which also contains
the intelligence required to compute the pilot path. The Actor requests the
MFDriver to 'perform' the Move and to follow the pilot path it computes. The
Actor may also wish to impose acceleration constraints upon the MFDriver.
For example, if the automatically guided vehicle is transporting fragile equip-
ment, then the automatically guided vehicle may not accelerate or deccelerate
too fast. The velocity profile is not expressive enough to govern acceleration
behavior, especially because this behavior depends upon traffic conditions
that are different every time. To this aim FoRcEs introduces the notion of a
Precaution, which imposes an acceleration constraint until it is removed by
the Actor. A Move is also equipped with the intelligence to compute the ref-
rence acceleration according to the velocity profile and the current velocity
observed by the MFDriver, and Precautions. The MFDriver can request a Move
to compute the pilot path and the reference acceleration via the following
interface.

```java
interface Move {
    ...  
    void update(FuSpace g, double Ug, double Vg);
    FuPath getPilotPath();
    double getReferenceAcceleration();
}
```

update(g,Ug,Vg): computes the pilot path and the reference acceleration \(a\)
according to the current observed functional state \(g\), depicted in figure
4.5 as \(g(t)\), the current observed evolution \(Ug\), and the current observed
velocity \(Vg\). These observations are made by the MFDriver via its sensors.

getPilotPath: returns the FuPath indicating the direction in which the MFDriver
must travel through its FuSpace.

ggetReferenceAcceleration: returns the current reference acceleration \(a\) indi-
cating how fast the MFDriver must accelerate.

computing the pilot path

The deviation, depicted in figure 4.5 as \(\alpha\), is computed by minimizing the
distance between the observed functional state \(g\) and \(f\). Because the the
arbitrary FuTrace may have a circular shape, it is also necessary to minimize
the difference between the evolution \(u(t)\) associated with \(f\) and the evolution
\(u(t - \delta t)\) determined during the previous computation of the deviation. The
computation of the deviation is simplified by using the sensor-based observed evolution \( u_g \). For an automatically guided vehicle the observed evolution may be is inferred using an odometer.

The direction of the pilot path is determined according to the pilot functional state, depicted in figure 4.5 as \( F(U_p) \). The pilot evolution \( U_p \) is computed to be \( u + d \cdot a + b \), where \( a > 0 \), and \( b > 1 \) are parameters set by the Actor (during construction of a Move). If \( U_p \) exceeds the domain of the FuTrace, then the extension of the FuTrace is used to compute the pilot functional state. Finally, the pilot path is the FuPath from \( (G(t)) \) through \( F(U_p) \).

computing the initial reference acceleration

The initial reference acceleration is the acceleration required to achieve the desired velocity. In general, the desired velocity is the velocity associated with the pilot functional state. However, the velocity profile associated with of the extension of a FuTrace is zero. In order to avoid the machine from stopping too early, the desired velocity is computed as follows. Suppose \( V \) is the FuTrace velocity profile, \( u(t) \) is the current evolution, \( U_p \) is the pilot evolution, and \( V_g \) is the current observed velocity \( V_g \). The pilot velocity \( V_p \) is computed to be maximum velocity \( \cdot V(U_p) \), where the maximum velocity is a parameter set by the Actor. The initial reference velocity \( V_a \) is computed to be maximum velocity \( \cdot V(u(t)) \). The reference velocity \( V_r \) is computed to be \( \gamma \cdot V_a + (1 - \gamma) \cdot V_p \), where \( 0 \leq \gamma \leq 1 \) is a parameter set by the Actor (during construction of a Move). Finally, the initial reference acceleration is computed to be \( (V_r^2 - V_g^2)/2(U_p - u(t)) \); this value may not exceed the maximum acceleration nor the maximum deceleration, which are also parameters set by the Actor (during construction of a Move).

computing the final reference acceleration

Suppose use of a shared resource \( R \) starts beyond a particular evolution value \( x \) along FuTrace \( F \). The Actor will wish to force the MFDriver to decelerate and stop at \( F(x) \), but only if \( R \) is not available. As soon as \( R \) is available, the Actor will inform the MFDriver that it may accelerate and continue on its way. The initial reference acceleration, which is computed according to the static velocity profile, is not expressive enough to describe the behavior that depends upon dynamic traffic conditions. FoRcEs introduces the Precaution to impose a minimal deceleration constraint upon the MFDriver until it is removed by the Actor. It is accessible via the following interface.
interface Precaution {
    void setMove(Move move);
    double getMinimalDeceleration()
    void remove();
}

setMove: sets the Move the Precaution is associated with.

getMinimalDeceleration: gets the current minimal deceleration constraint. The
deceleration may be computed using the current deviation, evolution,
and evolution velocity, as computed by the Move during its current in-
vocation of update.

remove: removes this Precaution, the minimal deceleration will no longer be
imposed. Only invoked by the Actor, informs the MFDriver that it may
accelerate and continue on its way.

The traffic control engineer may define a custom Precaution that computes a
minimal deceleration with respect to any values it can acquire via the Move
it is associated with. To this aim the Move interface also defines the following
methods.

interface Move {
    void update(FuSpace g, double gU, double gV);
    double getObservedEvolutionVelocity();
    double getEvolution();
    double getDeviation();
}

getObservedEvolutionVelocity: returns $gV$ passed via update.

getEvolution: returns the current evolution $u$ computed during update.

getDeviation: returns the current deviation $d$ computed during update.

FoRcEs predefines the BrakewayPrecaution, which specifies an evolution value
$x$ and a specified minimal deceleration $a$. It instructs the MFDriver to stop
before $F(x)$ at a deceleration rate of $a$, where evolution value $x$ is defined
relative to the beginning of the FuTrace $F$ contained within the Move. As
depicted in figure 4.6, the acceleration of an automatically guided vehicle is
constrained by a BrakewayPrecaution in such a way that the evolution veloc-
ity will remain below the solid curved line. At time $T_p$ the MFDriver must
start decelerating, this time depends upon the velocity of the automatically guided vehicle. Prior to this time, the BrakewayPrecaution will not impose any deceleration constraint. Several Precautions may be active at the same time, and the reference acceleration of a Move is computed to be the greatest lower bound of all acceleration constraints imposed by Precautions and the initial reference acceleration. To do this, a Move must be able to access its own Precautions, and this is possible via its own interface.

```java
interface Move {
    ...
    Precaution[] getPrecautions();
    Precaution getPrecaution(int i);
}
```

Precaution[] getPrecautions(): get all contained Precautions.

Precaution getPrecaution(int i): get the $i^{th}$ contained Precaution. Index 0 is reserved for the Precaution that forces the automatically guided vehicle to stop at the end of the Move's contained FuTrace. This Precaution is also referred to as the standard stop-at-end Precaution.

event-oriented interaction with the Actor using Flags

An Actor will wish to be notified whenever an event it is interested in is experienced by the MFDriver. In figure 4.7 it is depicted that use of a shared resource $R$ starts beyond evolution value $x$ along a FuTrace representing a path $P$, and the use stops beyond evolution value $y$. The Actor has associated a BrakewayPrecaution with the FuTrace forcing the MFDriver to decelerate and stop at $x$. If the automatically guided vehicle is ever to reach its destination the Actor must organize access to $R$ and remove the Precaution, and for this
Figure 4.7: Precaution and flags

reason the Actor wishes to be informed of the moment the Precaution starts imposing its minimal deceleration constraint. The Actor will also wish to know when the MFDriver has progressed beyond \( y \). FoRcEs introduces the Flag as a software component that will send a Letter back to the Actor whenever the event it defines occurs. It is accessible via the following interface.

```java
interface Flag {
    void setMove(Move move);
    void setLetter(Letter letter);
    void raise()
}
```

**setMove:** sets the Move the Flag is associated with.

**setLetter:** sets the Letter, it will be sent whenever the Flag is raised. The Actor will set a Letter addressed to itself.

**raise:** raises the Flag, if the event defined by the Flag has occurred. the current deviation, evolution, and evolution velocity, as computed by the Move during its current invocation of update. The occurrence of the event may be computed using the current deviation, evolution, and evolution velocity, as computed by the Move during its current invocation of update.

The traffic control engineer may define a custom Flag defining an event with respect to any values it can acquire via the Move it is associated with. To this aim the Flag may also use the Move methods used by the Precautions. FoRcEs predefined the BrakewayFlag which is intended for use together with a BrakewayPrecaution, as depicted in figure 4.7. It also specifies an evolution value \( x \) and a specified minimal deceleration \( a \), it is raised at the time a
BrakewayPrecaution with the same parameters will start imposing its minimal deceleration. In this way the Actor will be informed of the moment the MFDriver will start decelerating in order to stop in x. FoRcEs also predefines the EvolutionFlag, which specifies an evolution value y. Whenever the MFDriver progresses beyond y the Flag will be raised. As depicted in figure 4.7, an EvolutionFlag may be used to inform the Actor a resource is no longer required.

The Actor may associate a regular Letter with a Flag, but it may also be Sign or a Signal. FoRcEs defines two different software components for both the EvolutionFlag and the BrakewayFlag. The EvolutionFlagSign and the BrakewayFlagSign will notify the Actor via a Sign upon occurrence of the specified events. The EvolutionFlagSignal and the BrakewayFlagSignal will notify the Actor via a Signal upon occurrence of the specified events. A Move will invoke raise upon all its contained Flags whenever it is requested to update according to a new observed functional state. To do this a Move must be able to access the Flags, this is possible via its own interface.

```java

interface Move {

    ...
    Flag[] getFlags();
    Flag getFlag(int i);
}
```

Flag[] getFlags(): get all contained Flags.

Flag getFlag(int i): get the ith contained Flag. The Flag associated with index 0 is reserved for the Flag the indicating the MFDriver is finishing the Move.

**standard Precautions and Flags**

In general, an Actor will wish the MFDriver to stop at the end of a Move's FuTrace. For this reason FoRcEs introduces the following standard Precaution.

**stopAtEnd:** the MFDriver must stop at $F(U)$, the destination of the contained FuTrace. This Precaution is associated with index 0.

FoRcEs also introduces the following standard Flags.
finishing: the **MFDriver** is decelerating in order to stop at $F(U)$. This event is of interest to the **Actor** because it indicates that it must acquire permission for the **MFDriver** to perform the next **Move**. This **Flag** is associated with index 0.

started: the **MFDriver** has progressed beyond $F(0)$, the performance of the **Move** has started. For an automatically guided vehicle this event implies that it has started to travel to the destination of the path. This event is of interest to the **Actor** because it indicates that the first resource associated with the **Move** is indeed being utilized. This **Flag** is associated with index 1.

finished: the machine has arrived at $F(U)$. For an automatically guided vehicle this event implies that it is ready to exchange customer goods at its destination. This **Flag** is associated with index 2.

beyond: the machine has passed beyond $F(U + length)$. Assuming $F(U)$ corresponds with a parking area resource, this event implies that the tail of the automatically guided vehicle has moved beyond the parking area and the last resource required by this **Move** is is no longer being utilized. This **Flag** is associated with index 3.
the MoveIB software component

The procedure for following a FuTrace does not refer to XYASpace nor to any other particular function space, it is only related to the notions of a straight line and distance, which are defined by FuSpace. This implies the FuTrace following procedure is generic for any type of machine of which the function space is well-defined. The same is true for the computation of the reference acceleration according to the velocity profile and Precautions, and for the computations required for raising Flags. This allows FoRcEs to provide a default implementation for the FuTrace following procedure within a software component called MoveIB, it also defines fields for the standard Precaution and the standard Flags.

```java
class MoveIB implements Move {
    FuTrace fuTrace;
    Precaution stopAtEnd;
    Flag started, ready, finished, beyond;
    ...
    public Move(double a, double b, double maxVelocity,
                 double velocityGamma,
                 double maxAcceleration, double maxDeceleration
    )
    { ... }
    void update(FuSpace g, double Ug, double Vg) { ... }
    FuPath getPilotPath() { ... }
    double getReferenceAcceleration() { ... }
    ...
}
```

**Move**: constructs a new **Move**. Parameters \(a\) and \(b\) are required to compute the pilot state, parameter \(\text{velocityGamma}\) is required to compute the reference velocity and the initial reference acceleration, and parameters \(\text{maxAcceleration}\) and \(\text{maxDeceleration}\) are constraints upon the reference acceleration.

The **MoveIB** implements all methods defined by the **Move** interface as discussed above. The **MFDriver** only methods invokes getPilotPath and getPilotPath, and method setNext discussed below.

creating a Move

Consider the infrastructure depicted in figure 5.1 on page 102 of the next chapter. The path from A to E is subdivided into five shared resources
called A, B, C, D, and E. Suppose traveling from A to E is sub-divided into four moves called A2B, B2C, C2D, and D2E. The definition of B2C is presented below. The definition of the three other moves is practically identical and will not be discussed any further.

```java
public class B2C extends MoveIB {
    public B2C(AtoB aToB, XYASpace bLocale) {
        super(15,1,0.5,1.5,1.5);
        fuTrace = new CircularBendLeft(8, Math.PI/2, bLocale);
        double traceEnd = fuTrace.getEndEvolution();
        stopAtEnd = new BrakewayPrecaution(traceEnd, 0.9);
        started = new EvolutionFlagSign (0);
        finishing = new BrakewayFlagSign (traceEnd, 0.9);
        finished = new EvolutionFlagSign (traceEnd);
        beyond = new EvolutionFlagSignal(traceEnd+15,aToB,"beyond");
    }
}
```

B2C is defined as extensions of MoveIB because the FuTrace following procedure described at the beginning of this section is appropriate for following the contained CircularBendLeft. If it had not been appropriate, then a complete new FuTrace following procedure could have been defined within this class definition. It would have to be made available to the MDriver via the Move interface that this class would have to implement. In line 00 the constructor of MoveIB is invoked in order to set the parameters determining the trace following behavior. In order to compute a pilot functional state at approximately the front of the automatically guided vehicle, which is 15 meters long, the pilot parameter \( a \) is set to 7.5, assuming a FuTrace is defined with respect to the center of the automatically guided vehicle. It is assumed the automatically guided vehicle can follow the FuTrace very well; therefore the pilot parameter \( b \) is set to 1. The pilotGamma parameter is set at 0.5, in order to average the initial reference velocity and the pilot velocity. The maximum acceleration and deceleration rate are both set at 1.5 \( m/s^2 \), this value is rather small in order to avoid damaging fragile goods. In line 01 a CircularBendLeft is created as the B2C's FuTrace. The radius of the bend is 8 meters, the end angle is \( \pi/2 \), and the length of the bend as returned in line 02 is therefore \( 4\pi \). In line 03 a BrakewayPrecaution is created in order to force the MDriver to stop the automatically guided vehicle before it passes beyond the end of the CircularBendLeft. The deceleration rate is set to 0.9 in order to ensure a very calm behavior. The started-Flag created in line 04 will be raised whenever the automatically guided vehicle has passed by B, and the finished-Flag created in line 06 will be raised whenever the automatically guided vehicle
passes by C. The `finishing-Flag` created in line 05 will be raised whenever the automatically guided vehicle starts to decelerate in order to stop in C. After the `stopAtEnd` is removed, the `beyond-Flag` created in line 12 will be raised whenever the automatically guided vehicle no longer occupies C. Note that for A2B the `beyond-Flag` indicates the automatically guided vehicle no longer occupies B and another automatically guided vehicle is free to use B, which is an intersection. How the `Actor` may request the `MFDriver` to perform `Moves` and how the `Actor` may monitor the progress of the `Moves` is discussed in section 4.3.

### 4.3 The Machine Function Driver

The machine function driver is an `Agent` equipped with trace intelligence required to follow the pilot course computed by a `Move`, and having the dexterity required to avoid obstacles it can observe via its sensors. The `MFDriver` provides one service: performing a coherent sequence of `Moves` for an `Actor`, the service is accessible via the `MFDriver` interface presented below.

```java
interface MFDriver extends Service {
    void prepare(Move move) throws MoveIncoherentException;
    void begin (Move move) throws MoveIncoherentException;
    ...
}
```

**begin:** the pilot intelligence contained within the `Move` is available for computing the pilot path and reference acceleration. The contained `FuTrace` will be followed by manipulating the controls of the machine in the direction of the pilot state. Furthermore, obstacles will be avoided if observed.

-MoveIncoherentException: thrown if the `Move` is of an unknown type or if the contained `FuTrace` is incoherent with respect to the current `FuTrace`.

**prepare:** the `Move`'s contained `FuTrace` is available for computing the pilot state. The machine may not evolve within the domain of the contained `FuTrace`. This method is discussed further below.

-MoveIncoherentException: thrown if the `Move` is of an unknown type or if the contained `FuTrace` is incoherent with respect to the current `FuTrace`.

In section 4.2 it is discussed how a `FuTrace` may be followed by computing a pilot path according to a pilot state, which in turn is computed according to
a pilot evolution. The pilot evolution may be computed to be some distance beyond the domain of the FuTrace, for this reason every FuTrace defines a FuPath extension beyond its end. In general, the extension will not coincide with the FuTrace contained within the Move a MFDriver must follow next. This implies the pilot functional state may jump from one functional state in FuSpace to another when the MFDriver is requested to begin performing its next Move. This sudden jump of the pilot functional state may not be desirable, a possible result could be a jerking movement by the machine. In order to avoid this, the MFDriver allows the Actor to request it to prepare for a Move some time before it requests the MFDriver to begin performing the Move. Whenever the pilot evolution computed by a Move exceeds the domain of its contained FuTrace, then it will request the next Move to compute the pilot functional state, if it exists. To this aim the following two methods are defined in the Move interface and implemented by the MoveIB software component.

```java
interface Move {
    void setNext(Move move);
    FuSpace getPilotState(double Up);
}
```

**setNext**: sets the next Move, it must be performed after this Move. This method is invoked by the MFDriver upon this Move if it was the last previously accepted Move and the MFDriver has now accepted a new next Move via prepare or begin.

**getPilotState**: returns the pilot functional state. Suppose the domain of the contained FuTrace is \( d \) and the pilot evolution is \( U_p \). The value trace.getPoint(Up) is returned if \( U_p < d \). If not, and the next Move is set, then the next Move is requested to compute the pilot functional state using \( U_p - d \). Otherwise the pilot functional state is computed using the FuTrace extension.

Figure 4.8 depicts a sequence of Moves maintained by the MFDriver, the two Moves to the left have been passed via begin, the Move to the right has only been passed via prepare. After the MFDriver has produced a new observed state \( g = G(U_g) \), the MFDriver will select the begin-Move for which the observed evolution \( U_g \) falls within the domain of the contained FuTrace. The MFDriver may not select a prepare-Move. If \( U_g \) exceeds the domain of a Move's contained FuTrace then the Move no longer contains any relevant information and is discarded. The selected Move will be requested to update after which the MFDriver will request the computed pilot path and reference acceleration. During update
the pilot evolution $U_p$ required for computing the pilot functional state is computed as described above.

**monitoring the performance of a Move**

Recall the infrastructure depicted in figure 5.1 and the Move definition of B2C as presented in the previous section. The definitions of A2B, C2D and D2E are similar to that of B2C. The only difference is the shape of the contained F uTrace. Suppose the Actor wished the M FDriver to travel from A to E, then it must create the four required Moves as follows.

00 XYASpace aLocale = ...
01 a2b = new AtoB(aLocale);
02 XYASpace bLocale = a2b.fuTrace.getEndPoint();
03 b2c = new BtoC(bLocale);
04 XYASpace cLocale = b2c.fuTrace.getEndPoint();
05 c2d = new BtoC(cLocale);
06 XYASpace dLocale = c2d.fuTrace.getEndPoint();
07 d2e = new BtoC(dLocale);

After construction, the Actor may request the M FDriver to perform the Moves.

08 agvMfd.begin(a2b);
09 agvMfd.begin(b2c);
10 agvMfd.begin(c2d);
11 agvMfd.begin(d2e);

After requesting the M FDriver to perform the Moves, the Actor must monitor the progress of the Moves. The Actor must do so for at least two reasons: the first is that it must remove stopAtEnd precautions; the second is that it must
know when the automatically guided vehicle has arrived at its destination. Consider the following pseudo-script.

```plaintext
12 a2b.finishing.watch();
13 // organize permission to use C
14 a2b.stopAtEnd.remove();
15 b2c.finishing.watch();
16 // organize permission to use D
17 b2c.stopAtEnd.remove();
18 c2d.finishing.watch();
19 // organize permission to use E
20 c2d.stopAtEnd.remove();
21 d2e.finished.watch();
```

In lines 13, 16, and 19, the Actor must organize access to shared resources represented within the virtual infrastructure, how this is achieved is discussed in section 5.1. In lines 12, 15, and 18, the Actor watches for the moment the automatically guided vehicle must start decelerating in order stop at the end of a FuTrace contained within a Move. This implies the resource required to perform the next Move is required. After this succeeds the stopAtEnd Precautions are removed, preventing the resources from being used too early. In the last statement, the Actor watches for the arrival of the automatically guided vehicle in E, upon occurrence of this event customer goods may be exchanged.

Adding and removing the standard stopAtEnd Precaution imposes an extra burden upon the Actor, which may be delegated to the MFDriver. It seems natural that the Actor must acquire permission to use the first resource required by a Move before it requests the MFDriver to begin performing the Move. If so, the MFDriver may be configured to remove the Precaution associated with index 0, which is intended to be the standard stopAtEnd Precaution, of its last Move at the time it is requested to begin performing the next Move. Regarding the example above, a simplified pseudo-script could be as follows.

```plaintext
// organize permission to use B
agvMfd.begin(a2b);
agvMfd.begin(a2b);
// organize permission to use C
agvMfd.begin(b2c);
agvMfd.begin(c2d);
```
c2d.finishing.watch();
// organize permission to use E
agvMfd.begin(d2e);
d2e.finished.watch();

adapting the speed

The MFDriver also defines four methods via which the Actor may manipulate the reference acceleration as computed by the current Move.

    interface MFD extends Service {
        ...
        adaptSpeed(double rate);
        interruptExecution()
        resumeExecution()
        cancelExecution()
    }

adaptSpeed: sets a factor between 0 and 1 with which the reference velocity, which is determined via the velocity profile, must be multiplied.

interruptExecution: stores the current velocity profile multiplication factor and then sets it to be 0.

resumeExecution: resets the velocity profile multiplication factor to the value that was stored during the previous of interruptExecution.

cancelExecution: sets the factor with which the reference velocity is multiplied to 0 and disposes of the current Move when the velocity becomes 0.

machine control

FoReEs is also concerned with machine control and provides an architecture for the MFDriver. In figure 4.9, which is a function diagram, four Mfd sub-agents are depicted. The MFDriver continuously cycles through a control loop at a particular frequency, possibly between 5-50 Hz. In every cycle the MFDriver will request the sub-agents to perform their tasks in the order described below.
4.3. THE MACHINE FUNCTION DRIVER

**state finder:** responsible for observing the sensors, inferring the observed evolution \( U_9 \), inferring the observed functional state \( G(U_9) \), maintaining a **dynamic model of the environment**, and maintaining **machine response values**. The state finder is specific with respect to the machine type.

For an automatically guided vehicle, the state finder must infer the automatically guided vehicle's position and orientation by observing sensors. Within mini-world the mini-agv receives its functional state from a local positioning system at regular intervals. The updates are received via a wireless local area network, which may be viewed as an externally oriented sensor. During intervals internally oriented sensors measuring wheel angles and rotations are used to estimate its position and orientation. A model of the environment is inferred utilizing external sensor measurements, possibly infra-red distance measurements. Response values are sensor measurements, which indicate how the machine has responded to mechatronic settings. For the mini-agv, the wheel rotation sensors provide response data with respect to velocity settings.

**move driver:** responsible for **machine function** related computations, in particular the computation of the pilot path and the reference acceleration, according to Moves and the observed functional state. The move driver is generic and is the same for all types of machines. Indeed, the implementation of a move driver only refers to **Move, FuSpace, and FuTrace**.

The **MFDriver** will forward the **Moves** it receives via **begin or prepare to**
the move driver, including the requests to set the reference acceleration multiplication factor.

**instructor**: responsible for *machine type* related computations according to the pilot course. It is the instructor which is equipped with dexterity. In order to avoid obstacles, the instructor must determine its position and orientation with respect to a model of the environment provided by the state finder. Upon discovering an obstacle along the pilot course, the instructor must compute a minimum deceleration value in order to stop before a collision with the obstacle occurs. The desired acceleration is the greatest lower bound of the minimum deceleration value and the reference acceleration.

For an automatically guided vehicle, this means that the instructor must compute the direction in which it must steer its wheels with respect to the frame of the automatically guided vehicle and the *desired acceleration*. The instructor will produce *machine instructions* which are machine type specific.

**physical driver**: responsible for *machine instance* related computations according to the machine instructions. To this aim the physical driver maintains a *response model* of the machine which indicates how the machine is expected to respond to actuator settings. The machine instructions imply a desired machine response and the physical driver will compute new actuator settings according to the desired machine response and the response model. Machines of the same type may respond differently to machine control settings, and the response of a machine may change during its lifetime. Therefore the physical driver will continuously compare the actual machine response with the desired response and adjust the response model if necessary.

For an automatically guided vehicle, the response model indicates how the steering angle and acceleration change according to actuator settings. The acceleration response may change due to a variety or reasons: an increased load, or a new orientation of the wind. Furthermore, sensor observations from which response values are inferred may change: the sun may change odometer sensor observations by shining upon the air-filled wheels, or increased load has decreased wheel diameters.

Every sub-Agent, and the composing MFDriver-Agent, has access to AgileSpace and can therefore communicate with any other Agent within the logistic system if so required.
4.3. THE MACHINE FUNCTION DRIVER

MFDriver failure

If a machine, such as an automatically guided vehicle, suffers mechanical failure then the MFDriver will eventually discover that the machine no longer responds to control settings as expected. Eventually, the MFDriver will determine that the problem cannot be resolved. If so the MFDriver must notify its machine monitor and abort all the Signs and Signals contained within the Moves the Actor has requested it to perform. Consequently, the Actor will also be notified of the failure - an issue discussed in section 5. If the machine suffers electrical failure, then the computer platform embedded within the machine will fail abruptly and the MFDriver will not have the time to notify its machine monitor nor abort any of the Signs and Signals and the machine monitor must discover the failure at its own initiative. Because the Actor and the MFDriver both reside upon the computer platform embedded within the machine, the Actor will fail abruptly together with the MFDriver. The mechatronic interface of the machine should be constructed in such a way that electrical failure will result in an emergency stop. Unfortunately, unresolved programming errors would sometimes cause the MFDriver to fail after which it will stop passing new instructions to the mechatronic interface. In order to achieve fail safe behavior, the mechatronic interface should be programmed to full-stop whenever the MFDriver fails to issue actuator settings for too long.

Simulation

Recall figure 2.5 on page 32; the FoRcEs Agents, which are MFDdrivers, interact with external machine agents. Regarding the development of a logistic control system, the external agents are replaced by simulated counterparts until deployment. However, the notion of an external agent depends upon how the figure is interpreted with respect to a particular logistic system. For the mini-agv, one could think of the interface as the software driver to the mechatronic device controlling the wheels and the motors, or as the MFDriver interface presented to a TrAcEs Actor. Both views are discussed below.

The notion of a Move is very powerful within the context of simulation with respect to a TrAcEs Actor. The FoRcEs workbench provides GenericMFDriver, which can simulate the performance of Moves defined with respect to any type of FuSpace. For a machine that is not dexterous the GenericMFDriver will closely mimic the behavior of a ’real’ MFDriver as it follows FuTraces according to the associated velocity profile. Because Precautions are also defined with respect to FuSpace the velocity maintained by the GenericMFDriver is also constrained by Precautions. Finally, the GenericMFDriver will also raise Flags be-
cause these are also only defined with respect to FuSpace. The GenericMFDriver
is a powerful tool with which the behavior of the Actor can be tested, it has
been used during development of the crossover transport system, as dis-

cussed in section 7.4. At this time the GenericMFDriver only functions in
‘normal’ time. In order to facilitate faster simulations, an extension of the
GenericMFDriver should be developed which computes the next moment in
time a Flag must be raised, or a Precaution becomes active, according to the
current evolution velocity and reference acceleration. In this way simulation
may be implemented in an event-sceduled manner assuming the scheduler
will only trigger a new event after all other Threads within the system have
become suspended.

If the MFDriver of a machine is non-dextrous, then its behavior is not in-
fluenced by data produced by externally oriented sensors in order to avoid
obstacles. If so, it is relatively easy to develop an Agent mimicking the be-
havior of the machine’s mechatronic interface because the value of internally
oriented sensors can be computed according to the response model main-
tained by the MFDriver’s physical driver. The MFDriver’s state finder will be
‘fooled’ into computing an observed functional state which is identical to the
next desired functional state. A mechatronic simulator Agent was also de-
veloped for the mini-agv, and its simulated behavior mimics the behavior of the
mini-agv very well.

Suppose an automatically guided vehicle is equipped with an infra-red sen-
so that its dexterous MFDriver must utilize to autonomously maintain its
distance to a predecessor. The generic MFDriver simply does not use such
data, and a mechatronic simulator Agent cannot compute such data accord-
ing the the MFDriver’s response model. In order to mimic the behavior of a
dexterous MFDriver its external environment must be simulated, which is a
daunting task. However, the FoRcEs workbench provides a 3d-visualization
environment called Virtuality, it has been built using Java3d [43]. In mini-
world, Virtuality is used to render a 3d-view of all the machines active within
the material handling system. Within Virtuality, MFDriver’s manifest them-
selves as solids with a position and orientation, and it can be programmed to
compute collisions between solids as they move with respect to each other,
providing a very powerful tool to model physical interactions between ma-
chines. For example, autonomous distance maintenance may be achieved by
attaching a fan-shaped solid, invisible within the 3d-view, to the front of an
automatically guided vehicle which corresponds with the range of infra-red
sensors. A collision between the fan-shaped solid and another machine or
obstacle will generate an event that may be used to simulate the infra-red
sensor.
4.4 Validation

In order to validate FoRcEs, it must be shown that the Move does possess enough expressive power to control an complex machine, such as an automatically guided vehicle, and it must be shown that Moves are expressive enough use the layout of a complex traffic control system. In chapter 7 it is shown how the Move is used to instruct automatically guided vehicles to travel over the crossover-layout, which is considered a complex layout. How the Move is used to control an automatically guided vehicle is discussed below.

The dimensions of the mini-agv, depicted in figure 4.1, are approximately 1/20-th of the dimensions of an automatically guided vehicles used to transport 40-foot containers. However, the mini-agv is considered to be representative of any automatically guided vehicle. The mini-agv is equipped with a MFDriver which does indeed perform sequences of Moves. Any Move containing a FuTrace within XYASpace is considered coherent with respect to another. If the beginning of a next FuTrace is at another position and orientation than the end of the current FuTrace, the mini-agv simply travels to the beginning of the next FuTrace in a straight line, however far away it may be. This behavior emerges due to the generic trace following behavior programmed within Moves. Obviously, this behavior must be utilized with care. At this time, there are some problems regarding the velocity, it is not estimated very well. As a result, the Flags may be raised too early. This problem is not considered a problem with respect to the definition of FoRcEs, but a problem of the mini-agv.
Chapter 5

Distributed Traffic Control

In chapter 4 it is shown how a MFDriver may perform a sequence of Moves generated by an Actor; the performance of a Move requires one or more shared infrastructure resources. In order to avoid collisions access to such resources must be organized. It may be true the MFDriver is capable of organizing access to such resources by using externally oriented sensors. If not, then the Actor must do so using a model of the infrastructure within the virtual world. In this chapter it is shown how a traffic control engineer may construct a virtual infrastructure within a distributed traffic control system. In chapter 2 the notion of traffic intelligence is introduced; it is required to navigate through the virtual infrastructure without collisions, and to generate Moves that minimize velocity variations. In chapter 6 it is shown how traffic intelligence may be modelled in a hierarchy of Actions. Upon assignment of a job, an Actor must acquire the appropriate Action from the traffic control system, much in the same way a tourist may acquire a folder describing a touristic trip from the local tourist office. In sections 5.2 and 5.3 it is shown how an Actor must interact with its environment in order to acquire Actions. In order to perform the job, the Actor must activate the Action after which the automatically guided vehicle will travel to the job's destination.

Within a well-structured controlled environment the set of shared resources does not change very often. This allows the traffic control engineer to model such shared resources beforehand within the virtual world as Semaphores, which may also model non-physical shared resources such as traffic control processes. Semaphores are the subject of section 5.1. Within Actions, traffic control engineers program access to shared resources using Tickets that represent Semaphores within a distributed environment. Advanced traffic intelligence such as such anticipation, resources selection, and resource collection, is also programmed within Actions utilizing Tickets.
In general, an infrastructure may be subdivided into disjunct areas with a particular function, such as intersections and parking lots. Such a disjunct area is associated with its own sub-scene, and a set of sub-actions to navigate through the sub-scene. In order to navigate across sub-scenes, super-actions are required that 'activate' sub-actions in a sequential manner. In section 5.2 it is shown how such a hierarchical approach to modelling traffic intelligence allows the design of scalable distributed traffic control systems.

5.1 Semaphores

In order to model shared resources within the virtual world it must be clear which resource attributes must be modelled. A primary attribute of a shared resource is the number of agents that can use it at the same time. A shared tool, such as a hammer, can only be used by a single carpenter at the same time, but a shared infrastructure, such as a road, may be used by several motorists at the same time. The notion of capacity associates a shared resource with a positive number representing the number of agents which may utilize the resource concurrently. It may be so that different types of agents require different amounts of capacity in order to use a resource. For example, a car may require a single unit of road capacity, but a truck may require two units of road-capacity. An agent has exclusive possession of capacity during time it uses a resource. Assuming an agent does not consume a resource, then the capacity will become available to other agents after some time. Essential is that the amount of resource capacity utilized by a group of agents concurrently may never exceed the capacity of the resource itself.

The notion of capacity is not restricted to infrastructure resources, it is also applicable within a more abstract context. For example, suppose the traffic is governed by the rule that vehicles approaching from the right have the right of way. If four vehicles arrive at an intersection the same time, then none will have the right of way. The access procedure for utilizing the intersection is a shared resources with a capacity of less than four; the exact capacity also depends on the other rules that govern traffic.

In order to model a shared resource within the virtual world it is necessary to design a suitable software component. The problem of organizing access to shared resources is well known in computer science, concurrent programs compete for access to memory, processors, and other devices such as printers. To solve this problem Dykstra introduced the semaphore to guard resource capacity by synchronizing concurrent programs [3]. For example, the single unit of capacity of a printer may be guarded by a semaphore called printer.
In order to print documents a program must first invoke \( P(\text{printer}) \). The operating system procedure call \( P \) is such that its invocation will only terminate after the \text{printer} capacity is available. After all documents are printed the program must invoke \( V(\text{printer}) \) indicating to the operating system the \text{printer} capacity is available once again. If a program is already using the printer then any other program invoking \( P(\text{printer}) \) will be de-activated until the program using the printer invokes \( V(\text{printer}) \), in this way at most one program will ever possess the printers capacity at any time. The TrAces \text{Actor} is a program responsible for governing the MFDriver controlling a machine, and in this chapter it is shown how collisions between machines may be avoided by synchronizing \text{Actors} via semaphores representing infrastructure resources. This is not as straightforward as it may seem because an automatically guided vehicle is subject to inertia and velocity variations must be avoided. The \text{Actor}, as a program, is de-activated and re-activated instantaneously during its interactions with a semaphore, and therefore the interaction must take the inertia of an automatically guided vehicle into account. Access to infrastructure resources may require prioritized traffic rules. For example, an ambulance should be allowed to cross an intersection first. TrAces introduces the \text{Semaphore} as a software component responsible for guarding the capacity of a shared resource used by TrAces \text{Actors}, it extends the original notion of a semaphore with prioritized traffic rules.

**designing virtual infrastructures**

A layout is a description of how the environment should be used, it is designed according to logistic insight. For a transport system, the layout describes routes that transport vehicles may follow. In general, the traffic control engineer will sub-divide the environment into disjunct areas according to the layout. At first sub-divisions will be coarse, identifying larger structures such as industrial facilities. The traffic engineer will continue sub-dividing the environment into smaller disjunct areas such as intersections and parking lots. An infrastructure resource is simply an area the traffic engineer does not wish to sub-divide any further and will represent in a model of the infrastructure.

In general, most shared resources are best represented by a \text{BasicSemaphore} that enforces the most basic of all traffic rules: \textit{first-come, first-serve}: resource capacity is assigned to the \text{Actor} that arrives first or waiting for the longest time. For example, consider the layout depicted in figure 5.1: a curved lane is intersected by two other lanes. It seems best to sub-divide the lane into five segments, each segment is smaller than an automatically guided vehicle and can only harbor a single automatically guided vehicle at
any time. Assuming the first Actor that wants to utilize a road segment may do so, then each of these five infrastructure resources may be guarded by a BasicSemaphore created with a single initial capacity. The five semaphores are created by the code presented below. Semaphores are created within the context of a Scene, this is a TrAcEs traffic control agent introduced in section 5.2. Note that the road segment and the Semaphores are only associated with each other in a conceptual manner, the traffic control engineer is responsible for using a Semaphore in its intended manner. This imposes quite a burden on the traffic control engineer, especially if the Semaphores model abstract entities other than shared infrastructure resources.

```java
sA = new BasicSemaphore(1);
sB = new BasicSemaphore(1);
sC = new BasicSemaphore(1);
sD = new BasicSemaphore(1);
sE = new BasicSemaphore(1);
```

The PrimeTicket

The original semaphore introduced by Dykstra, is accessed directly by a computer program via operating system calls p and v. Within a TrAcEs traffic control system an Actor is remote with respect to a Semaphore and direct access is simply not possible. To this aim TrAcEs introduces the PrimeTicket which represents a Semaphore within the context of an Actor, it presents the following interface to an Actor.

```java
class Ticket {
    boolean attempt();
    void reserve();
    void insist();
    void free();
```
5.1. SEMAPHORES

```java
int snip();
...
}
```

boolean attempt: returns true if the claimed resource capacity is granted immediately, false otherwise and the resource capacity is not claimed. The use of attempt is discussed further in section 6.1.

void reserve: requests the Semaphore to reserve resource capacity, without blocking. At the time the resource must be utilized the Actor must verify the claim is granted via invocation of insist. The use of reserve is discussed further in section 6.1.

void insist: requests the Semaphore to grant resource capacity as soon as possible, the invocation will block until the claimed capacity is granted.

void free: returns previously granted resource capacity to the Semaphore.

int snip: returns 0 if the Semaphore has not yet granted resource capacity. The use of snip suggests a busy-waiting programming strategy and is not advised from a concurrent programming point of view. It is not discussed further in this thesis.

The PrimeTicket provides access to a shared resource within a distributed environment. In section 6.1 the SelectTicket is introduced that may be used for selecting the first available resource. In section 6.1 the CollectTicket is also introduced that may be used for collecting a set of resources in an atomic manner.

**interlacement of PrimeTicket invocations**

Suppose an automatically guided vehicle is in B and must travel to D, as depicted in figure 6.2. First, the Semaphores guarding B, C, and D, must be requested to create a PrimeTicket representing a claim upon a single unit of capacity, as follows.

```java
tB = sB.createPrimeTicket(1);
tC = sC.createPrimeTicket(1);
tD = sD.createPrimeTicket(1);
```
Suppose an automatically guided vehicle is in B and must travel to D via C, as depicted in figure 5.2. The Actor must insist upon a PrimeTicket providing access to a road segment before the front of the automatically guided vehicle enters the shared resource, and the Actor must free the PrimeTicket after the tail of the of the automatically guided vehicle has passed beyond the road segment. This implies the invocations of insist and free upon PrimeTickets are interlaced: tc.insist in invoked before tb.free and tc.free is invoked after td.insist. This interlacement is essential in order to avoid collisions but also introduces difficulties regarding construction of scripts describing how the Actor must navigate through the infrastructure. In chapter 6 it is shown how to model traffic intelligence in such a way that the following script, for traveling from B to D via C, is safe and efficient.

\[ bToC.execute(); // insists and frees tc \]
\[ cToD.execute(); // insists and frees tD \]

**advanced traffic rules**

A Semaphore is created with an initial amount of capacity that represents the capacity of the shared resource it must guard. During its lifetime, Actors will request the Semaphore to create PrimeTickets via which they may acquire permission to utilize the resource it guards. It may occur that a traffic monitor, which may be a person responsible for the functioning of an automated transport system, decides that a particular resource is temporarily unavailable. If so the traffic monitor may decrement the amount of capacity the Semaphore may grant to Actors.

It may occur that several Actors are waiting for the Semaphore to grant resource capacity because it is depleted, if so traffic rules must determine which Actor will be granted resource capacity first. For example, suppose a single-lane bridge crosses a wide river from north to south. At most 15 automatically guided vehicles may cross the bridge at the same time, and no more than 10 automatically guided vehicles may cross the bridge in one direction if other
automatically guided vehicles are waiting on the other side. The Semaphore guarding access to the bridge must be created with an initial amount of 15 units of capacity, but a first-come first-serve traffic rule is inappropriate. TrAcEs introduces the AccessLine to allow the definition of complex traffic rules, an AccessLine represents the reason why an Actor requires resource capacity. Whenever an Actor requests a Semaphore to create a PrimeTicket it must also specify the AccessLine the PrimeTicket must be associated with whenever resource capacity is requested via reserve or insist. The Semaphore will assign capacity to the PrimeTicket at the front of the AccessLine with the highest priority, after which the PrimeTicket is pulled from the AccessLine. TrAcEs introduces the following four types of AccessLines that order their PrimeTickets differently.

```java
class FIFO extends AccessLine: first in first out, a queue.
class SCF extends AccessLine: smallest capacity request first.
class LIFO extends AccessLine: last in first out, a stack.
class RANDOM extends AccessLine: random.
```

TrAcEs introduces four types of Semaphores that enforce different traffic rules according to which the priority of AccessLines is determined.

**BasicSemaphore:** a Semaphore enforcing the most basic of all traffic rules: first-come, first-serve: The BasicSemaphore is equipped with a single FIFO AccessLine. The Actor need not refer to the internal AccessLine upon requesting a PrimeTicket. Whenever the BasicSemaphore has capacity available, it will assign capacity to the PrimeTicket at the front of the FIFO AccessLine.

**RankedSemaphore:** a Semaphore equipped with $N$ AccessLines. AccessLine number 1 has the highest priority, the $n^{th}$ the lowest. The RankedSemaphore will assign capacity to the PrimeTicket at the front of the highest ranked AccessLine first.

**CyclicSemaphore:** a Semaphore equipped with $N$ AccessLines, each is associated with a convoy-length. The AccessLines are queued, the AccessLine at the front of the cyclic-queue has the highest priority. Priorities are changed by pulling an AccessLine from the front of the cyclic-queue and pushing it into the end of the cyclic-queue. The convoy length is the maximum number of PrimeTicket assignments the AccessLine may
retain its highest priority at run-time. Priorities are also changed if the AccessLine at the front of the cyclic-queue becomes empty and there is a non-empty AccessLine elsewhere in the cyclic-queue.

CyclicExclusiveSemaphore: a Semaphore similar to the CyclicSemaphore, the difference is that an AccessLine is only pulled from the front of the cyclic-queue and pushed into the back of the cyclic-queue after all capacity assigned to the PrimeTicket on the AccessLine has been returned.

Recall the example of a single-lane bridge crossing a wide river from north to south. At most 15 automatically guided vehicles may cross the bridge at the same time, and no more than 10 automatically guided vehicles may cross the bridge in one direction if other automatically guided vehicles are waiting on the other side. The Semaphore guarding the bridge must be created with an initial capacity of 15 and two access lines, one for each direction and a convoy length of 10 for both. Whenever an Actor intends to cross the bridge, then it must request a PrimeTicket and indicate the direction it intends to cross the bridge by specifying the appropriate AccessLine. The Semaphore must be cyclic in order to alternate the direction in which traffic may flow, and the Semaphore must be exclusive in order to avoid deadlocks: the bridge has only a single lane, an automatically guided vehicle may not start driving to the other side before all automatically guided vehicles driving in the other direction actually arrive at the other side of the bridge.

```java
Semaphore sBridge = new CyclicExclusiveSemaphore(15,
        new FIFO("NorthToSouth", 10),
        new FIFO("SouthToNorth", 10));
```

Suppose an automatically guided vehicle must travel from the southern side of the bridge to the northern side. The Actor must request sBridge to create a PrimeTicket for a single capacity via "SouthToNorth".

```java
Ticket tBridge = sBridge.createPrimeTicket(1, "SouthToNorth");
```

At this time, only the BasicSemaphore is provided by the TrAcEs workbench.

## 5.2 Distributed Scenes

An ordered set of Semaphores guarding shared resources spawns a 'virtual infrastructure' called a scene. Figure 5.3 depicts the Semaphores guarding the
5.2. DISTRIBUTED SCENES

Figure 5.3: a hierarchy of Scenes

infrastructure introduced in section 5.1. In order to perform a job, the Actor must activate a program modelling the traffic intelligence to do so. Within this section it is shown how the Actor may acquire and activate such a program. From an object-oriented programming point of view with respect to the traffic control system as a whole, the program modelling traffic intelligence is not of primary interest, but the Object within which it is contained. TrAcEs introduces the Action as an Object that in intended to contain the traffic intelligence. For activation the Action presents the following interface to the Actor.

```
interface Action {
    ...
    void assimilate(MFDriver mfd, Actor actor, ...) ... ;
    void execute() ... ;
}
```

Any Object implementing this interface is called an Action. The complete definition of interface Action is presented in section 6.1.

void assimilate: sets the MFDriver that will perform the Moves this Action will generate upon activation, and the Actor that requested activation.

void execute(): activates the Action, it will generate a sequence of Moves for the MFDriver to perform. The invocation will block until the MFDriver is finishing the last Move.

During activation an Action will insist upon PrimeTickets in order to acquire permission to use resources required to perform the Moves it generates, and it will free PrimeTickets after such resources are no longer needed. An Action is
deactivated after the traffic intelligence it contains is no longer required. Unfortunately, this issue is not as straightforward as it seems due to PrimeTicket method invocation interlacements. This issue is discussed in detail in sections 6.1 and 6.2. However, within this section the reader may simply assume that termination of execute means the automatically guided vehicle has arrived at its destination, PrimeTickets have been insisted and freed at the right time, and the Action has been deactivated. How traffic intelligence must be modelled within Actions in order to achieve this result is the subject of the next chapter.

Actions and the scene are complementary; if the structure of the scene is modified then the Actions may be adjusted accordingly, for this reason Actions are maintained together with the scene by a traffic control Agent called a Scene. However, the traffic control engineer may choose to sub-divide the infrastructure into disjunct areas with a particular function. Such a sub-scene is associated with a set of sub-Actions to navigate through the sub-scene. In order to navigate across sub-scenes, super-Actions are required that ‘activate’ sub-Actions in a sequential manner. This suggests a hierarchal approach to traffic control. Figure 5.3 depicts a hierarchy of Actions maintained within a hierarchy of Scenes. Whenever the Actor intends to travel through a virtual infrastructure it must first acquire the appropriate Action from the Scene via the following interface.

```java
interface Scene ...
{
    ...
    Action createAction(String actionName, ServiceID actorID)
        throws ActionException;
}
```

createAction(String actionName, ServiceID actorID): creates an Action with the specified actionName for an Actor with the specified ServiceID.

`ActionException`: the Scene can not provide the Action, the `message` of the exception will indicate why. If the `message` equals "unknown" the Scene does not maintain an Action with the specified actionName.

Suppose an automatically guided vehicle is in A and its Actor intends to navigate E. Assuming the Actor is aware of the Scene-hierarchy controlling the traffic in that area and that it requires Action "AtoE" from abcdeScene, then it must execute the code presented below. Upon completion of line 03 the automatically guided vehicle has arrived in E.
Figure 5.4: Request `abcdescene` to create a new `AtoE`

01 aToE = abcdescene createAction("AtoE", actor.getServiceID());
02 aToE.assimilate(agtActor, agvMFDriver);
03 aToE.execute();

The **Actor** is a traffic control **Agent** that resides upon a machine together with its **MFDriver**. This means an **Actor** is remote with respect to **Scenes** that reside within the infrastructure. Why **Actors** and **Scenes** are distributed in this way is discussed in section 8.2. Because **Scenes** and **Actors** are remote with respect to each other, this raises the question where the **Action** is created: within the context of the **Scene**, or within the context of the **Actor**? In section 8.2 it is concluded it is best that the **Action** is created within the context of a **Scene** and **downloaded** to the context of an **Actor**. Fortunately, java-rmi provides technology that makes it is easy to download **Actions**, and any sub-**Object** it may contain, such as sub-**Actions** and **PrimeTickets**. For example, during statement line 01 above **Action** `AtoE` is created within the context of `abcdescene` and downloaded to the **Actor**. Downloading of an **Action** requires communication bandwidth; it may be that downloading a large **Action-tree** may take a long time. The traffic control engineer must therefore decide if an **Action** must be downloaded together with sub-**Actions**, or if sub-**Actions** should be downloaded during assimilation or activation.

Consider downloading of the **Action-tree** of `AtoE`. Assuming `abcdescene`, `bcsScene`, and `desScene`, all reside upon a different computer, then it is best that `AtoE`, `AtoC`, and `CtoE`, are downloaded to the **Actor** directly from their respective computers. This implies an `AtoE` should be downloaded without any sub-**Actions**, and that these sub-**Actions** should be downloaded during assimilation, or during activation, of an `AtoE`. This in turn implies that `AtoE` must be downloaded together with references to `bcsScene` and `desScene`. How this is achieved is shown below. First the partial definition of `abcdescene` is presented.
Figure 5.5: assimilate is invoked upon the downloaded AtoE

class ABCDEScene implements Scene {
    BasicSemaphore sA;
    Scene bcScene;
    Scene deScene;
    ...
    public Action createAction(String actionName, ServiceID actorID) {
        if (actionName.equals("AtoE")) {
            return new AtoE(this); // construct a new AtoE
        }
        ...
    }
}

Agent abcdeScene maintains references to its two sub-Scenes. How these references are acquired is beyond the scope of this thesis, but these may have been passed to abcdeScene during its construction. Figure 5.4 depicts invocation of createAction("AtoE", actorID), this occurs during line 01 above. Upon such an invocation abcdeScene creates a new AtoE, the constructor is defined as follows.

public class AtoE implements Action {
    Scene bcScene, deScene;
    ...
    public AtoE(ABCDEScene abcdeScene) { // constructor
        this.bcScene = abcdeScene.bcScene;
        this.deScene = abcdeScene.deScene;
    }
}

During construction a new AtoE is passed references to the two sub-Scenes, after these references are stored the new AtoE is downloaded to the Actor.
Figure 5.6: aToC is downloaded together with aToB and cToC

Figure 5.5 depicts the result of `createAction("AtoE",actorID)`, the new AtoE has been downloaded to the Actor, together with its stored references to the sub-Scenes. Next, line 02 is executed and aToE is requested to assimilate. This method is defined as follows.

```java
public class AtoE implements Action {
    Scene bcScene, deScene;
    Action aToE, cToE;
    ...
    public void assimilate(MFDriver mfd, Actor actor, Action super) {
        aToC = bcScene.createAction("AtoC", actor.getServiceID());
        cToE = deScene.createAction("CtoE", actor.getServiceID());
        aToC.assimilate(mfd, this);
        cToE.assimilate(mfd, this);
    }
}
```

Figure 5.5 depicts the invocation of the first statement of `aToE.assimilate`, bcScene is requested to create an AtoC. Because sub-Actions AtoB and BtoC are also maintained by bcScene, and therefore reside upon the same computer, these may be downloaded together. Figure 5.6 depicts the result of `bcScene.createAction("AtoC", actorID)`, the new AtoC has been downloaded to the Actor, together with its sub-Actions. Figure 5.7 depicts the result of line 02 above, the complete AtoE Action-tree has been downloaded. Next, line 03 is executed and aToE is requested to activate. This method is implemented as follows.
public class AtoE implements Action {
    ...
    Action aToE, cToE;
    public void execute() {
        ...
        aToC.execute();
        cToE.execute();
    }
}

the wrong reference to a sub-Scene

Suppose abcdeScene has acquired the wrong reference to bcScene. If so, then it will occur that the wrong Scene will be requested to create a sub-Action called "AtoC". If by coincidence the wrong Scene also provides an Action called "AtoC", then the wrong sub-Action will be downloaded to the Actor that refers to a layout located at an arbitrary location. It activation of the wrong Action does occur, then the MFDriver will be requested to perform a Move that contains a FuTrace at an arbitrary location. Obviously, this is a dangerous situation and the MFDriver should refuse to perform the wrong Move because it is not coherent. For this reason it is best that a Move is only considered coherent if its begin-point equals the end-point of its predecessor. If the issue of Move-coherence is ignored, then it seems unlikely such a fault will not be discovered before deployment of the traffic control system.

an arbitrary scene hierarchy

The notion of a hierarchy of Scene Agents seems to introduce a problem, neighboring shared resources within the infrastructure may only have a common
super-Scene in the top of the Scene hierarchy. It may not be so that an Actor must stop the machine function driver at a particular location because of a virtual barrier imposed upon the Actor by an Scene hierarchy which is arbitrary from the Actor’s point of view. Fortunately, this problem is similar to the problem of avoiding velocity variations. An automatically guided vehicle need only stop before using a shared resource if it is already being used by another automatically guided vehicle. In section 6.2 it is shown how this issue is resolved by modelling traffic intelligence in such a way that the Actor will start organizing access to a shared resource before the automatically guided vehicle must start decelerating. If permission to use the resource is granted immediately then the automatically guided vehicle need not decelerate at all. The same anticipatory behavior may be used to download, assimilate, and activate, an Action before the automatically guided vehicle must start to decelerate for a virtual barrier between two geographical areas guarded by different Scenes.

Scene failure

Failure of a Scene may not compromise traffic safety. It is assumed all Semaphores maintained by the Scene fail together with a Scene because they reside together within the same context. PrimeTickets are programmed to display robust and fail-safe behavior and will never allow an invocation of insist to succeed without explicit assignment of capacity by the Semaphore. This issue is discussed in section 8.3. Fail safe behavior also implies the MDFDriver may not be requested to begin performing a Move for which permission to use the requires resources has not been acquired via PrimeTickets first. Because such behavior is defined within Actions downloaded from Scenes, all responsibility regarding the design of a fail-safe traffic control system lies with traffic control engineers developing Scenes.

implementation of Scene Agents

Recall the Scene hierarchy depicted in figure 5.3. The bcsScene maintains two BasicSemaphores in order to guard road segments B and C, and an Action AtoC that refers to two sub-Actions AtoB and BtoC. The bcsScene is an instance of class bcsScene, defined as follows.
class BCscene implements Scene {
    Transform locale;
    ServiceID serviceID;
    Semaphore sB;
    Semaphore sC;
    ...
    public BCscene(Transform bLocale, ServiceID serviceID) ...
    Action createAction(String actionName, ServiceID actorID) ...
}

BCscene(Transform bLocale, ServiceID serviceID): constructs a new BCscene with respect to the locale of b. In lines 02 and 03 below the single capacity BasicSemaphores guarding the resources B and C are created. In line 04 the Scene registers itself with AgileSpace after which it can be found by Actors.

00 this.locale = bLocale;
01 this.serviceID = serviceID;
02 this.sB = new BasicSemaphore(1);
03 this.sC = new BasicSemaphore(1);
04 AgileSpace.registerService(this, serviceID);

Action createAction(String actionName, ServiceID actorID): creates an AtoC for an Actor if actionName equals "AtoC". Creation of AtoC is discussed below.

A Scene is created by the traffic monitor, probably via the command-line. The traffic monitor must do so before any automatically guided vehicles attempt to enter the infrastructure the Scene must guard. The traffic monitor knows where segment B is located within the environment and what the Scene's ServiceID must be. The traffic monitor knows these facts after these have been asserted by the traffic control engineer during design of the traffic control system. How these two individuals communicate is beyond the scope of this thesis, but during the development stage the traffic control engineer and the traffic monitor may be the same person. The traffic control engineer may choose to define any number of other creation parameters deemed useful, such parameters may determine the shape or the size of the geographical area the Scene will guard, or its traffic control behavior. The CrossoverScene, the case study of this thesis discussed in chapter 7, demonstrates the use of such creation parameters to set the shape and the size of the geographical area it will guard.
Before an Actor can request a Scene to provide it with an Action, it must first acquire a reference to the Scene via AgileSpace. Assuming the Actor is aware of bcScene's ServiceID it can acquire an AtоО by executing the three statements presented below. An Actor may be informed of a Scene's ServiceIDs in two ways. The most common way is via the traffic intelligence contained within Actions acquired from a super-Scene. For example, Agent abcedeScene is passed the ServiceIDs of sub-Scenes bcScene and deScene by the traffic monitor upon creation. These ServiceIDs will be passed to any AtоО abcedeScene will create for an Actor, in this way the Actor also acquires the ServiceIDs of bcScene and deScene. Note that this approach is not possible for the Scene providing a top-level Action corresponding with a job. In this case the job description itself must provide the ServiceID. Assignment of jobs is discussed further in section 5.3.

ServiceID bcServiceID = ...
Scene bcScene = AgileSpace.lookupService(Scene,bcServiceID);
Action aToC = bcScene.createAction("AtoC");

Agent abcedeScene maintains a single BasicSemaphore in order to guard segment A, and provides Actions EnterA and AtoE, its definition is also in many ways similar to that of BCscene and is not discussed any further. Agent deScene is practically identical to bcScene, they only differ with respect to the shape of the layout and the locale within the environment, and its class definition is practically identical to BCscene. In general, a traffic control engineer may identify recurring structures within a layout. If so a Scene designed to govern traffic within a single occurrence of a recurring structure will be suitable for governing traffic in all occurrences of the structure. The only difference between the Scenes will be their locale and their ServiceID, their class definitions will be identical. Re-using Scene class definitions is a very powerful design strategy, significantly shortening the traffic control system development cycle.

scalable distributed traffic control systems

As an automatically guided vehicle travels through the environment, it will be utilizing at least one shared resource represented within the virtual infrastructure. In general, an Actor will communicate with only a few Semaphores at the same time, and a Semaphore will only communicate with a few Actors at the same time that are in the vicinity of the resource. The exact number depends upon the detail with which the infrastructure is modelled, and
the speed of the automatically guided vehicles. However, there will be a practical limit to this number because the logistic engineer will not identify more shared resources within an environment than is considered useful, and automatically guided vehicles will not travel at extreme speeds. This implies that an individual Semaphore does not impose 'heavy' requirements upon the processing power and communication bandwidth of the computer network. A TrAcEs traffic control system may consist of a large number of Semaphores, if all Semaphores were to reside upon the same computer then the processing power of the computer and the communication bandwidth with the Actors may not suffice. Because Semaphores do not communicate with each other they may easily be distributed over multiple computers, solving the issue of limited processing power without imposing extra requirements upon communication bandwidth.

The infrastructure is subdivided into disjunct areas with a particular use, such as intersections and parking lots, and guarded by a hierarchy of Scenes, as discussed above. In general Semaphores will be maintained by 'low-level' Scenes. It seems natural to host a low-level Scene upon a computer 'near' to the guarded infrastructure because then communication requirements for interaction with Actors will be localized. Assuming local area networks, including the wireless sections, will be able to provide the required communication bandwidth the issue of scalability with respect to the number of infrastructure elements which can be guarded via a virtual infrastructure is solved.

The Actor will also download Actions from a Scene via the computer network. In general, it will only do so if it is in the geographical vicinity of the area guarded by the Scene. For low-level Scenes this implies downloading of Actions will only impose localized communication requirements, and it is assumed the local area network can meet these requirements. For higher-level Scenes the guarded geographical area may be very large and it may not be clear where the Scene should be hosted. In general, a high-level Scene will only need to communicate with Actors intending to migrate between geographical areas governed by sub-Scenes. Assuming not all Actors intend to migrate at the same time then the communication bandwidth required by a super-Scene will not be extreme, and probably similar to that of a sub-Scene. Assuming the local area networks possess enough surplus communication bandwidth to facilitate communications between Actors and Scenes geographically far apart, then the geographical location of a super-Scene is not a problem.

The discussion above supports the conclusion that the TrAcEs traffic control design method is scalable, assuming the traffic control engineer designs
a hierarchy of Scenes, and assuming the hierarchy of Scenes is mapped onto
a computer network in a sensible manner. It is also assumed that the net-
work can provide the required communication bandwidth between Scenes and
Actors, and that the network can provide the required processing power. The
explosive growth of the internet has demonstrated that this is not an issue.

5.3 Actor

The Actor is a traffic control Agent that can perform a coherent sequence of
jobs. In this section it is shown an Actor may be assigned a job, and how the
Actor may download the Action containing the traffic intelligence required to
perform the job. The Actor interface is defined, in part, as follows.

```java
interface Actor ... {
    ...
    void perform(Job job, Service service) throws Exception;
    Service getService() { ... }
    ...
    class Job implements ExceptionListener {
        ...
        ServiceID sceneID;
        String actionClassName;
    }
}
```

perform: accepts a job to perform.

Exception: thrown if the job is not compatible with the Service, or with
the previous job.

getService: returns the Service associated with the current job. This method
may be invoked by an activated sub-Action in order to interact with the
Service-Agent for which the current job is being performed.

TrAcEs provides the ActorIB software component for rapid development of
Actors, it is generic for all types of Jobs and MFDivers. For every Job, it
executes code similar to the four statements presented below. In line 00
the Actor acquires a reference to a Scene via AgileSpace using the ServiceID
provided by the Job. In the next statement the Actor downloads an Action
using the class name provided by the Job, and passing its own ServiceID. In
line 03 the Actor assimilates the Action, upon termination of execute in line
04 the Job is completed.
01 Scene currentScene = AgileSpace.lookup(Scene, job.sceneID);
02 Action currentAction =
02 currentScene.createAction(job.actionClassName, this.serviceID);
03 currentAction.assimilate(this, this.mfDriver);
04 currentAction.execute();

a sequence of jobs

Upon termination of execute upon a job-Action, the Actor will check for any new Jobs passed to it via perform. If there is a next Job to perform, the Actor will execute the sequence of statements above again with respect to the next Job. If not, the Actor must advertise itself via AgileSpace. In this advertisement it must be clear where the Actor is in order for potential LoGoS clients to know which Jobs it can perform. An Actor could advertise the current functional state within its FuSpace but this does not seem practical. To this aim LoGoS introduces the notion of a LogisticPosition, which is intended to represent the origin and destination of a Job, such as a transshipment area or a parking lot. Logistic engineers request traffic control engineers to design Actions which instruct an automatically guided vehicle to travel from one LogisticPosition to another, for this reason the Action interface also defines the following two methods.

```java
interface Action {
    ...
    LogisticPosition getOrigin();
    LogisticPosition getDestination();
}
```

Upon termination of execute upon a job-Action, the Actor will request its current LogisticPosition by invoking getDestination on the job-Action. If there is no next Job to perform the Actor will advertise its Service with its current LogisticPosition as an attribute. If there is a next Job then the LogisticPosition-origin of the next Job must equal the LogisticPosition-destination of the current Job. If these are not equal, the Actor will throw a JobIncoherentException during invocation of perform(job).

the Actors lifetime

The lifetime of an Agent is the time during which it performs Jobs using the same ServiceID. During this time an Agent may die, possibly due to a com-
puter platform failure, and respawn, any number of times. Upon respawning an Actor must progress through two phases: the booting phase, during which it advertises itself with AgileSpace for the first time, and the operational phase, during which it actually performs Jobs. In general, an Actor is associated with the same MFDriver and the same machine during its lifetime. Both Agents reside upon the computer embedded within the automatically guided vehicle, both die whenever the embedded computer fails, and both are respawned whenever the embedded computer is re-booted. The Actor can not assume the embedded computer is re-booted while the automatically guided vehicle is in a safe location, its first priority is to ensure it is in a safe place. To do this, the Actor first request the MFDriver for its functional state within its FuSpace. Next, the Actor requests AgileSpace to provide it with a reference to an arbitrary Scene. It can do so be requesting for a SceneService without specifying a particular ServiceID. This arbitrary Scene will be requested to provide a reference to the Scene that governs the traffic in the local area where the Actor has found itself. The local Scene will be requested to provide an Action which will guide the automatically guided vehicle to a safe LogisticPosition upon activation. To this aim all Scenes implement the following methods.

```java
interface Scene ... {
...
Scene getScene(FuSpace machineLocale);
Action createAction(FuSpace machineLocale, ServiceID actorID);
}
```

Scene getScene(FuSpace machineLocale): returns a reference to the Scene governing traffic in the vicinity of the machineLocale.

Action createAction(FuSpace machineLocale, ServiceID actorID): returns a new Action containing the traffic intelligence required to park the automatically guided vehicle at a safe LogisticPosition.

For example, suppose an automatically guided vehicle was started near location A guarded by abcdeScene. The MFDriver will discover its locale upon respawning, suppose this locale is called nearA. Suppose the Actor arbitrarily locates deScene via AgileSpace, upon invocation of deScene.getScene(nearA) a reference to abcdeScene will be returned. Action "EnterA" will be downloaded to the Actor upon invocation of abcdeScene.createAction(nearA, actorID), this Action contains the traffic intelligence to instruct the automatically guided vehicle to move from any locale to A. After termination of enterA.execute
the Actor will have acquired the capacity of resource A and the automatically guided vehicle will be in A. The Actor will register its Services with AgileSpace, its LogisticPosition attribute will be set to enterA.getDestination, which is A. Other Agents may 'see' the newly registered Actor and know it can perform transport jobs originating in A. Assuming a transport system Agent is aware of abcdeScene it may request the Actor to perform a job corresponding with "AtoE".

**Actor failure**

The Actor may fail due to a programming error. In general such a programming error will be found in an Action downloaded from a Scene. In section 6.3 it is discussed how an Action may be programmed to deal with failures, and how a persistent failure may be propagated to the Actor via the Action-tree. A persistent failure will result in an Exception being thrown from currentAction.execute, after which the Actor will invoke job.handle, passing the Exception. The Actor may also fail abruptly, together with the MFDriver, upon failure of the computer platform embedded within the handling machine. If so, the Actor will not have the opportunity to inform the Actor-monitor of its failure and the monitor must discover the failure upon its own initiative.
Chapter 6

Traffic Intelligence

In the previous chapter it is shown how a hierarchy of Scenes maintaining Semaphores and Actions may constitute a distributed traffic control system. In this chapter it is shown how traffic intelligence may be modelled in a hierarchy of Actions. In chapter 5 the Action is introduced from the perspective of activation, but it may be required to access an Action before and after activation, and it may be required to monitor Action-events during activation. To this aim the Action interface also defines other methods; its complete definition is as follows.

```java
interface Action ...
{
    void assimilate(MFDriver mfd, Actor actor) ...
    Ticket getAccessTicket(int i) ...
    void prepare() ...
    void execute() ...
    void execute(Ticket[] entryTickets) ...
    Ticket[] getExitTickets() ...
    void run() ...
    Sign getSign(int i) ...
}
```

**assimilate:** assimilates the traffic knowledge contained within this Action with the Actor, after which the Actor knows how to navigate through the virtual infrastructure. Assimilation is introduced on page 107.

**Ticket getAccessTicket(int i):** returns a reference to the Ticket providing access to the $i^{th}$ resource required to perform Moves. Access-Tickets are used within the context of anticipation; this is discussed on page 124.
prepare: prepares this Action for activation; discussed on page 132.

execute(): activates this Action; invocation will block until the MFDriver is finishing the last Move. Activation is introduced on page 107.

execute(Ticket[] entryTickets): activates this Action and the invocation will block until the MFDriver is finishing the last Move. The entry-Tickets are available to the traffic intelligence contained within the Action, and are used within the context of anticipation and de-activation; this is discussed on pages 125 and 131.

Ticket[] getExitTickets(): returns the exit-Tickets of this Action; used within the context of deactivation: this is discussed on 131.

void run: activates this Action in an concurrent manner, allowing the super-Action to perform some other activity at the same time; this is discussed on 132.

Sign getSign(int i): returns a Sign according to the index. Signs are used in order to monitor the progress of an Action after (concurrent) activation; this is discussed on page 133.

Recall the infrastructure depicted in figure 5.1, the layout defines a route from A to E, and B is an intersection. Figure 5.7 depicts a hierarchy of Scenes that guard that infrastructure. An AtoB will instruct the MFDriver to travel to the intersection, and a BtoC will instruct the MFDriver to travel from the intersection. Both sub-Actions refer to B, so which of the two is responsible for organizing access to it? It seems natural that AtoB must organize access to its destination, which is B. This suggests an Action is responsible for organizing access to resources its Move requires, except the first resource, because access to the first resource is organized by the previous Action.

Figure 6.1 depicts traveling from A to E. From an object-oriented programming point of view, the interactions with a particular PrimeTicket should be contained within a single Action-Object. This implies that the Action that obtained a PrimeTicket from a Semaphore, must also insist and free the PrimeTicket. Figure 6.1 depicts how PrimeTicket method invocations are interlaced as the automatically guided vehicle progresses from A to E, and it seems impossible to maintain an object-oriented approach to modelling Actions. In section 6.2 it is shown how concurrent programming techniques may be used to model the interaction between Actions and PrimeTickets in an object-oriented manner. As a result, the traffic intelligence for traveling from A to E may be programmed in an intuitive script as follows.
6.1. ADVANCED TRAFFIC INTELLIGENCE

Figure 6.1: using Actions & ticket interlacement

aToB.execute(); // obtains, insists, and frees tB
bToC.execute(); // obtains, insists, and frees tC
cToD.execute(); // obtains, insists, and frees tD
dToE.execute(); // obtains, insists, and frees tE

It would be detrimental to the performance of a transport system if an automatically guided vehicle must stop every time execute terminates. In section 6.2 it is shown how the interaction between an Actions and Moves may be modelled in order to avoid velocity variations. As a result, the script above is not only intuitive, it is also efficient. In section 6.1 it is shown how to model route selection, anticipation, resource collection, and interaction with external Agents. This section also demonstrates how deadlocks, which are related to collecting resources, may be avoided. During the time an Action is active it will generate events, in section 6.3 it is shown how an Action's events may be defined and monitored.

6.1 Advanced Traffic Intelligence

During the performance of a job, an Actor will wish to select a route, anticipate upon arrival in congested areas, and collect infrastructure resources. Regarding resource collection, the Actor must be cautious not to deadlock with other Actors trying to collect the same resources. Furthermore, the Actor may wish to perform some other activity during the time a sub-Action is active, such as interact with external Agents, or monitor the progress of the sub-Action. In this section it is shown how traffic intelligence may be modelled in order to implement the behavior described above. The ActionIB software component is introduced, it may be extended by traffic control en-
engineers with traffic intelligence, and implements standard behavior that a traffic control engineer may consider useful.

**anticipation**

Figure 6.2 depicts the invocation of `insist` and `free` upon `tc` after activation of `bToC`. The `Actor` may wish to anticipate upon the activation of `bToC` by invoking `tc.reserve` some time before `bToC.execute`. In order to do so the `Actor` must acquire a reference to `tc` from `bToC`. To this aim, the `Action` interface defines the `getAccessTicket` method.

**Ticket getAccessTicket(int i):** returns a reference to the `Ticket` providing access to the `i`th required shared resource. The resource an access-`Ticket` provides access to must be described by the programming interface documentation of the `Action-class` implementing this method. This method may only be invoked before activation.

`Action BtoC` will return the internally defined `Ticket tc` upon invocation of `getAccessTicket(0)`. The code implementing this behavior is presented in section 6.2. Consider the following `Action-script`, the `Actor` anticipates upon the activation of `bToC` by reserving `tc` before activating `aToB`. It may be true `tc` has already acquired access to intersection B before `bToC` is activated, and the automatically guided vehicle may continue at full speed.

```java
Ticket tc = bToC.getAccessTicket(0);
tc.reserve();
aToB.execute();
bToC.execute();
```

The invocation of `aToB.execute` may take a long time, especially if the distance between `A` and `B` is large. Reserving upon `tc` before `aToB.execute` may therefore be too early, which is detrimental to the performance of the transport
system: other automatically guided vehicles may not cross over intersection B without a good reason. It should be possible to reserve upon tc some time during the activation of aToB. To this aim the Action interface defines a variant of the execute method which allows the Actor to pass Tickets to an Action upon activation.

\[
\text{execute(Ticket[]}[], \text{entryTickets}) : \text{activates this Action and the invocation will block until the MFDriver is finishing the last Move. The entry-Tickets are available to the traffic intelligence contained within the Action. How the Action will interact with the Ticket, and the timing of this interaction, must be described by its programming interface of the Action-class implementing this method.}
\]

With respect to the example above, Action aToB could be programmed to invoke reserve upon the single entry-Ticket half-way during the activity. If so, tc.reserve will be invoked within the following Action-script when the automatically guided vehicle is half-way from A to B.

\[
\text{Ticket tc = bToC.getAccessTicket();}
\text{aToB.execute(tc);}
\text{bToC.execute();}
\]

In general, the use of reserve will be detrimental to the performance of the traffic control system, because automatically guided vehicles will acquire permission to use resource before it is actually required. On the other hand, it may be desired that a particular automatically guided vehicle, such as a rescue vehicle, may be allowed to progress faster than regular automatically guided vehicles.

resourced (route) selection

Figure 6.4 depicts two possible paths from west to east, via the north or the south. In order to select between the northern route or the southern route the Actor may use the attempt method. Consider the following Action-script.

\[
\text{tN = westToNorth.getAccessTicket();}
\text{if (tN.attempt())} \{
\text{westToNorth.execute();}
\} \text{else} \{
\text{westToSouth.execute();}
\}
\]
CHAPTER 6. TRAFFIC INTELLIGENCE

Figure 6.3: choice via attempt

In the selection example above the Actor has a preference for traveling along the northern route. It order to select a route according to the first available resource TrAcEs introduces the SelectTicket, it is created with a set of sub-Tickets, and implements the following methods.

SelectTicket(Ticket[] subTickets): constructs a new SelectTicket, the sub-Tickets may only be PrimeTickets; arbitrary Ticket nesting is not available at this time.

boolean attempt: returns true if at least one invocation of attempt upon the sub-Tickets return true, false otherwise. Method attempt in invoked upon the sub-Tickets in the order in which these were declared, the first to return true is the selected Ticket.

void reserve: invokes reserve upon the sub-Ticket that will acquire permission to use its resource first. How this non-deterministic behavior is emulated is shown in section 8.5.

void insist: invokes insist upon the sub-Ticket that will acquire permission to use its resource first, the invocation will block until the claimed capacity is granted. After the claim is granted the sub-Ticket is the selected Ticket.

void free: invokes free upon the selected sub-Ticket.

int getSelectedlndex: returns the index of the selected sub-Ticket, -1 otherwise.

Consider the Action-script presented below. in lines 00 and 01 references to the Tickets providing access to the northern and southern routes are acquired.
6.1. **ADVANCED TRAFFIC INTELLIGENCE**

Figure 6.4: collecting Tickets

In line 02 the `SelectTicket` is created, and in line 03 the selected sub-Ticket is determined according to which route is available first. In lines 04 to 07 the Action for traveling along the northern or southern route is activated, according to the selected sub-Ticket. These is no need to free the selected sub-Ticket because the `westToNorth` or `westToSouth` are responsible for freeing their own Tickets.

```
00  Ticket tN = westToNorth.getAccessTicket();
01  Ticket tS = westToSouth.getAccessTicket();
02  SelectTicket tNorS = new SelectTicket(tN,tS);
03  tNorS.insist();
04  switch(sT.getSelectedIndex()) {
05      case 0 : westToNorth.execute(); // tN has index 0 in line 02
06      case 1 : westToSouth.execute(); // tS has index 1 in line 02
07  }
```

Nondeterministic selection occurs in all concurrent programs because one cannot predict the speed with which a Thread will execute code. In Java, Threads may wait upon notification by another Thread. If a group of Threads is waiting and notification occurs, it can not be predicted which Thread will be notified first [39]. As a result, a Thread may wait indefinitely. The difference with a SelectTicket is that it will select the first available resource, a resource will not be ignored. The non-deterministic aspect is that it is not known which resource will be available first.

**collecting (infrastructure) resources**

An activity may require a set of resources. For example, an automatically guided vehicle must travel from the north-west to the south-east, an activity modelled by an Action called `NorthWestToSouthEast`. The lane that the automatically guided vehicle must follow intersects with another lane connecting
the south-west with the north-east. Figure 6.4 depicts how the intersection is guarded by two Semaphores. Within NorthWestToSouthEast a PrimeTicket tN and tS must be created via Semaphores sN and sS, respectively. It seems valid to insist upon these Tickets is the order in which these are required. Suppose another Action SouthWestToNorthEast models traveling from the south-west to the north-east, the same reasoning would lead to the following two Action-scripts.

```
NorthWestToSouthEast               SouthWestToNorthEast
01  tN = sN.createPrimeTicket(1);  ||  tS = sS.createPrimeTicket(1);
02  tS = sS.createPrimeTicket(1);  ||  tN = sN.createPrimeTicket(1);
03  sN.insist();                   ||  tS.insist();
04  sS.insist();                   ||  tN.insist();
```

Concurrent activation of a NorthWestToSouthEast and a SouthWestToNorthEast may deadlock in line 04, if both scripts execute line 03 at approximately the same time. This deadlock could be resolved by imposing an order in which resources may be claimed. For example, the traffic control engineers may agree to only claim resources from north to south. Unfortunately, this imposes an extra burden upon traffic control engineers designing these Actions. Furthermore, even if such an order is imposed then deadlocks may still occur if traffic control engineers wish to anticipate upon utilizing the resources by invoking reserve before insist. Consider the following two Action-scripts.

```
NorthWestToSouthEast               SouthWestToNorthEast
01  tN.reserve();                   ||  tN.reserve();
02  tS.reserve();                   ||  tS.reserve();
03  tN.insist();                    ||  tN.insist();
04  tS.insist();                    ||  tS.insist();
```

Concurrent activation of a NorthWestToSouthEast and a SouthWestToNorthEast may deadlock in line 04, if both scripts execute line 01 at the same time. Deadlock can be avoided if lines 01 and 02 are always executed in a single haul. This may be achieved by using a third Semaphore. Consider the following two Action-scripts that use an extra semaphore called sx. In line 03 tx created via sx, and in line 04 tx is insisted. During concurrent activation of both Actions, only one of the two may proceed beyond this statement line 04. As a result, only one of the two will reserve tN and tS in a single haul before freeing tx in line 07.
NorthWestToSouthEast  SouthWestToNorthEast

01 tN = sN.createPrimeTicket(1);  ||  tS = sS.createPrimeTicket(1);
02 tS = sS.createPrimeTicket(1);  ||  tN = sN.createPrimeTicket(1);
03 tX = sX.createPrimeTicket(1);  ||  tN = sN.createPrimeTicket(1);
04 tX.insist();  ||  tX.insist();
05 tN.reserve();  ||  tN.reserve();
06 tS.reserve();  ||  tS.reserve();
07 tX.free();  ||  tX.free();
08 sN.insist();  ||  tS.insist();
08 sS.insist();  ||  tN.insist();

This solution demonstrates how a Semaphore may be used to guard an abstract resource: the traffic control process governing the use of the intersection. Unfortunately, this solution imposes an extra burden upon traffic control engineers: they must agree upon the existence and use of sX. It would be preferred if traffic control engineers could do their work independently, if possible. To this aim the CollectTicket is defined, it is created with a set of sub-Tickets. It implements the following methods.

CollectTicket(Ticket[] subTickets): constructs a new CollectTicket, the sub-Tickets may only be PrimeTickets, arbitrary Ticket nesting is not available at this time.

boolean attempt: returns true if all invocations of attempt upon all the sub-Tickets return true, false otherwise. If any invocation of attempt upon a sub-Ticket returns false, then all sub-Tickets are freed.

void reserve: invokes reserve upon all the sub-Ticket in an atomic manner. How this atomic behavior is achieved is discussed in section 8.5.

void insist: invokes reserve upon itself after which it invokes insist upon all the sub-Tickets, the invocation will block until all invocations of insist terminate.

void free: invokes free upon all the sub-Tickets.

Consider the following two Action-scripts, for which there is no danger of deadlock. The two traffic control engineers designing these script must still be aware of each others work, but less agreements must be made between them.
LeftToMerge     ||     RightToMerge
||
ct = new CollectTicket(tN,tS);    ||    ct = new CollectTicket(tS,tN);
ct.reserve();    ||    ct.reserve();
// tN and tS both reserved    ||    // tN and tS both reserved
ct.insist();    ||    ct.insist();
// tN and tS both insisted    ||    // tN and tS both insisted

unstructured interaction with other Agents

The TrAcEs architecture imposes a structure upon the environment of an Action: a Scene created the Action, an Actor activated the Action, and a Service-Agent requested the Actor to activate the Action in order to perform a job. An Action may wish to interact with these Agents, or other Agents it is aware of, using Letters. For example, it may not be known at the time of activation if the automatically guided vehicle must travel via a northern route of via a southern route. Assuming the Action is created with a reference to an external routing Agent, then it can retrieve a Letter containing the decision, via AgileSpace, some time after activation. Consider the following Action-script, in line 02 a BooleanLetter, sent by an external routing Agent, is retrieved from AgileSpace. In line 04 the content of the BooleanLetter is used to select the route to follow.

00  ServiceID externalAgent = ...
01  instruction = new BooleanLetter(externalAgent,this,null);
02  instruction = (BooleanLetter)AgileSpace.read(instruction);
03  boolean goWestToNorth = instruction.getValue();
04  if (goWestToNorth) {
05      westToNorth.execute();
06  } else {
07      westToSouth.execute();
08  }

In order to simplify programming of unstructured interaction via Letters the ActionIB software component, introduced below on page 135, provides references to Agents that will exist within the context of an Action, and several methods for sending and receiving Letters. However, the author does not prefer the use of Letters and advises the use of standard distributed programming techniques such as java-rmi.
retrieving exit-Tickets after de-activation

Recall the Ticket method invocation interlacement problem discussed at the beginning of this section. If concurrent programming is not used then it must be possible to extract a PrimeTicket from an Action after termination of execute. Suppose aToB.execute terminates upon arrival of the automatically guided vehicle in B. Before arrival PrimeTicket tB providing access to B must have been insisted. The automatically guided vehicle will depart to the next destination C upon invocation of bToC.execute, at this time aToB can not already have freed tB because the automatically guided vehicle still is in B. This will only be possible some time after invocation of bToC.execute when the automatically guided vehicle has traveled away from B. This means Ticket tB must be passed from aToB to bToC after execute terminates. To do this the Action interface defines the following method.

Ticket[] getExitTickets(): returns the exit-Tickets. The resources the exit-Tickets provide access to, and when these Tickets should be freed, must be described in the programming interface documentation of the Action-class implementing this method.

A Ticket extracted from a de-activated Action may be passed to the next Action as an entry-Ticket. Consider the following code.

```java
    aToB = new AtoB(sB);  // creates ticket tB
    aToB.execute(...);  // tB is insisted
    tB = aToB.getExitTickets()[0];
    bToC = new BtoC(sC);  // creates ticket tC
    bToC.execute(tB);  // tC is insisted, after which tB is freed.
```

The code above does not seem complicated or difficult to maintain. This is because it is assumed the automatically guided vehicle travels at a low velocity. In section 6.2 it is shown that the number of Tickets that must be passed between two successive Actions depends upon the velocity of the automatically guided vehicle. This implies that the simple code presented above would become complex and difficult to maintain if the automatically guided vehicle were allowed to travel at higher velocities. For this reason the use of exit-Tickets is not advised by the author because it can be avoiding using concurrent programming techniques, this is demonstrated in section 6.2.
preparing an Action via prepare

In section 4.3 it is shown how the Actor may avoid undesirable jerking movement of the machine by requesting the MFDriver to prepare a Move some time before performing the Move. If a sequence of Moves is generated by a single Action then it can be programmed to request the MFDriver to prepare for Moves it will generate in the future. But if a Move is generated by the next Action, the next Action must somehow be informed that it must request the MFDriver to prepare for Moves. To communicate such a request the Action interface defines the prepare method.

void prepare(): prepares this Action for activation. This method may only be invoked after assimilation and before activation. The precise semantics of prepare must be described by the programming interface documentation of the Action-class implementing this method.

In general, prepare implies the Action forwards the request to the first sub-Action it intends to activate, which in turn will do the same, and finally the first lowest-level sub-Action to be activated will request the MFDriver to prepare for a Move.

Preparing Actions has not been used to this date. The primary reason for this is that the only operational MFDriver is for the mini-agv, and within mini-world a jerking movement of a wheel is not considered a problem. Furthermore, the author is not convinced that preparing the MFDriver to perform Moves is the best approach for dealing with a sudden change of the pilot state. It may be better to program the MFDriver to enforce a ‘slow’ transition from the old pilot state to the new pilot state via a FuPath between the two states. Only after the transition is complete may the MFDriver may actually start performing the next Move.

concurrent Action activation

In section 5.2 the execute method is introduced and defined to activate an Action and to block until the MFDriver is finishing the last Move. It may be a problem for the traffic control engineer designing a super-Action that invocation of execute will block and that the Thread cannot be used to perform some other activity, such as monitoring other events the Action may generate. For this reason the Action interface provides the following methods for concurrent activation of an Action that is non-blocking.
6.1. ADVANCED TRAFFIC INTELLIGENCE

void run(): activates this Action in an concurrent manner with its own private Action-Thread.

void run(Ticket entryTicket): activates this Action in a concurrent manner with its own private Action-Thread. How the Action interacts with the entry-Ticket must be described in the programming interface documentation of the Action-class implementing this method.

void run(Ticket[] entryTickets): activates this Action in an concurrent manner with its own private Action-Thread. How the Action interacts with the entry-Tickets must be described in the programming interface documentation of the Action-class implementing this method.

Suppose an Actor wishes to interact with an external Agent during the time a sub-Action is active. A possible implementation of such an interaction is defined in the following code.

```
AtoC.run();
Letter letter = new StringLetter(this, externalAgent, "HELLO");
AgileSpace.send(letter);
Letter reply = new StringLetter(this, externalAgent, "GOODBYE");
AgileSpace.receive(reply);
...
cToE.run();
```

Using run introduces the possibility of activating an Action before the finishing-event of its predecessor occurs. If so, the MDriver may be requested to perform a Move too early. To avoid this, a super-Action is required to monitor the finishing-event of a sub-Action before activation the next sub-Action. The finishing-event may be accessed via the following method.

Sign getSign(int i): returns a Sign according to the index. Index 0 is reserved for the Sign via which the Action’s finishing-event can be monitored. The events which may be monitored via the Signs returned by this method must be described in the programming interface documentation of the Action-class implementing this method.

Consider the following code, during line 02 and line 05 the Actor-Thread has the opportunity to deploy some other activity. In order to avoid activating Actions too soon the finishing event of the concurrently activated Actions is explicitly monitored.
Monitoring of getSign(0) upon concurrent activation of an Action via run imposes an extra burden upon the traffic control engineer who must remember to do so. It is possible to program the Action-tree in such a way that a super-Action prevents a sub-Action from activating before getSign(0) of its predecessor has occurred. This possibility is discussed on page 137.

In the introduction of this chapter it is stated that concurrent programming techniques allow sequence of sub-Actions to be active concurrently with respect to each other. These techniques are also used to implement concurrent activation, as discussed in this section. However, the issue is different: concurrent activation refers to concurrency with respect to an Action and its super-Action. The use of run is not advised by the author because the traffic control engineer should use the concurrent programming techniques directly. From a concurrent programming point of view, it is best a Thread is used for one particular activity and should not be used for all kinds of other activities. Regarding the Action-script, an alternative implementation is as follows.

```java
firstOtherThread = new Thread(){
    public void run() { firstOtherActivity() } }  
firstOtherThread.start();
aToC.execute();
firstOtherThread.join();
// aToC and firstOtherActivity are both complete

secondOtherThread = new Thread(){
    public void run() { secondOtherActivity() } }  
secondOtherThread.start();
cToE.execute();
secondOtherThread.join();
// cToE and secondOtherActivityB are both complete
```
6.1. ADVANCED TRAFFIC INTELLIGENCE

standard Action events

The finishing-event is one of four events considered standard for any Action, these are defined as follows.

finishing: the standard finishing event of the last Move generated by the Action has occurred. This event may be monitored via getSign(0).

started: the standard started event of the first Move generated by the Action has occurred. This event may be monitored via getSign(1).

finished: the standard finished event of the last Move generated by the Action has occurred. This event may be monitored via getSign(2).

beyond: the standard beyond event of the last Move generated by the Action has occurred. This event may be monitored via getSign(3).

The traffic control engineer may define custom Action-events. These events must be described by the programming interface documentation of the Action-class. In section 6.3 it is discussed how the traffic control engineer may define events, how these may be monitored, and how events may be propagated.

the ActionIB software component

TrAcE provides the ActionIB software component for rapid development of Actions. It implements the Action interface and is suitable for downloading from a Scene to an Actor. This class is defined, in part, as follows.

```java
public abstract class ActionIB implements Action {
    ... 
    Sign started, finishing, finished, beyond;
    public void run() { ... }
    public void execute() { ... }
    protected abstract void script();
    protected void setSign(int i, Sign sign) { ... }
}
```

run: activates this Action in a concurrent manner with its own private Action-Thread, it is used to invoke script.

execute: activates this Action, the invocation will block until the MFDriver is finishing the last Move. This method is implemented as follows.
this.run();
this.getSign(0).watch();

script: to be overloaded with traffic intelligence.

setSign(int i, Sign sign) : set the Sign to be returned upon invocation of
getSign(i). It is intended that the Sign returned via getSign(0) is set
via this method before activation of an ActionIB.

An ActionIB is also equipped with a set of methods that simplify sending and
receiving of Letters via the AgileFrames communication space.

public class ActionIB implements Action {
    ...
    ServiceID serviceID;
    void send(Object dstAgent, Object content, String mark, int number)
    void send(Object dstAgent, Object content, String mark)
    void send(Object dstAgent, Object content)
    Object read(Object srcAgent, String mark, int number)
    Object read(Object srcAgent, String mark)
    Object read(Object srcAgent)
    Object readLast( Object srcAgent, String mark)
    boolean isLetter(Object srcAgent, String mark, Integer number)
}

send(Object dstAgent, Object content, String mark, int number): sends a Letter
containing content to the dstAgent with the specified mailMark and number.
The Thread is not blocked.

send(Object dstAgent, Object content, String mark): sends a Letter containing
content to the dstAgent with the specified mailMark. The number is irrele-
vant. The Thread is not blocked.

send(Object dstAgent, Object content): sends a Letter containing content to
the dstAgent. The Thread is not blocked.

read(Object srcAgent, String mark, int number): retrieves the content of a Let-
ter from the srcAgent with the specified mark and number. The Thread is
blocked until such a Letter arrives.

read(Object srcAgent, String mark): retrieves the content of a Letter from the
srcAgent with the specified mark. The Thread is blocked until such a
Letter arrives.
read(Object srcAgent): retrieves the content of a Letter from the srcAgent. The Thread is blocked until such a Letter arrives.

readLast(Object srcAgent, String mark): retrieves the content of the last Letter from the srcAgent with the specified mark. The mark may be null. The Thread is blocked until such a Letter arrives.

isLetter(Object srcAgent, String mark, Integer number): returns true if a letter from the srcAgent with the specified mark and number has arrived. The mark and the number may be null. The Thread is not blocked.

An Action, which resides within the context of an Actor after downloading and assimilation, may wish to interact with the Scene it originated from, the Actor that activated it, the Service-Agent that assigned the Actor the job for which this Action has been activated, or with its super-Action. To do this, the Action needs the ‘address’ of these Agents. To this aim the ActionIB defines the following fields.

```java
public class ActionIB implements Action {
    ...
    ServiceID actorID;   // Actor actor;
    ServiceID sceneID;   // Scene scene;
    Action superAction;
}
```

A reference to the Agent that assigned the Actor the current job may be acquired by invoking getService upon actor. An Action may also wish to interact with its sub-Actions, these are referenced by their name as declared within the particular Action.

**automatic monitoring of getSign(0)**

Upon concurrent activation of a sub-Action via run the next sub-Action may only be activated after getSign(0) of the previous sub-Action has occurred. An ActionIB could be programmed to enforce this behavior. Suppose all Actions within an Action-tree are ActionIBs. Whenever run is invoked upon a sub-Action its Thread first invokes superAction.wait(this) which will block until getSign(0) of the previous Action has occurred. Assuming the super-Action has activated a preceding sub-Action, this predecessor will have invoked superAction.wait(this) upon the super-Action first. This means the successor’s getSign(0) can be watched before returning from superAction.wait(this).
If the super-Action has not activated a preceding sub-Action, then invocation of superAction.wait(this) must be blocked until the super-Action itself has obtained permission from the super-super-Action to activate concurrently. If no super-super-Action exists, the super-Action is the root of the Action-tree. A root-Action will only be activated by the Actor via execute, and for this reason it may allow the first sub-Action to activate immediately.

an arbitrary activation signature

The traffic control Agent may wish to define an Action which requires special activation parameters other than the entry-Tickets. Arbitrary signatures for execute and run may be defined upon the class implementing the Action interface. These non-standard signatures must be described in the application programming interface of the class.

6.2 Basic Traffic Control

During performance of a job, an automatically guided vehicle must avoid collisions and velocity variations. In order to avoid collisions, an Action must insist and free PrimeTickets. In this section it is shown how concurrent programming techniques may be used to maintain an object-oriented approach to modelling the interaction between Actions and PrimeTickets. It is also shown how to model the interaction between Actions and Moves in order to avoid velocity variations.
Suppose it was not possible to allow actions to be active concurrently, and that termination of execute corresponds with de-activation of an action. If so, then an action would not be able to free its last prime ticket after execute terminates, because the automatically guided vehicle is still using the destination resource. This means the prime ticket must retrieved as an exit-ticket after execute terminates, and passed to the next action for freeing. Figure 6.5 depicts an automatically guided vehicle that must travel from B to D via C, each infrastructure resource is larger than the automatically guided vehicle. The depicted action-tree contains the traffic intelligence, it uses exit-tickets. Upon invocation of btoc.execute(tc) sub-action btoc is activated and tc is passed as an entry-ticket. Upon activation btoc insists upon tc before requesting the agv-mfdriver to perform b2c. Upon occurrence of b2c.finished the automatically guided vehicle has traveled from B and arrived in C. At this time the tail for the automatically guided vehicle must have passed beyond B because C is larger than the automatically guided vehicle, and it is safe to free entry-ticket tc. After tc.free, the invocation of btoc.execute(tc) terminates and the next action may be activated. Because tc has not yet been freed, it is extracted from btoc as an exit-ticket, and passed to ctoD as an entry-ticket. Action btoc will free tc upon arrival in D.

Within the super-action btoc.execute(tc) terminates upon arrival in C. The agv-mfdriver will also stop the automatically guided vehicle upon arrival in C because it has not been requested to perform the next move. This implies ctoD.execute(tc) will only be invoked after the automatically guided vehicle has stopped in C, and that the automatically guided vehicle will stop in C even if there are no other automatically guided vehicles using D. In order to avoid needless deceleration with respect to D ctoD.execute(tc) should be invoked upon occurrence of b2c.finishing instead of b2c.finished. If so, then ctoD invokes td.insist at the moment the agv-mfdriver must start decelerating in order to stop in C, instead of the moment the agv-mfdriver has stopped in C. If td.insist() succeeds immediately then the agv-mfdriver will be requested to begin performing c2d when the automatically guided vehicle is still traveling at full speed. If not, then the agv-mfdriver will decelerate the automatically guided vehicle in order to avoid a collision in D. As soon as td.insist() succeeds and the mfdriver is requested to begin performing c2d, then it will accelerate the automatically guided vehicle, even if it has not yet come to a full stop.

It is not safe to simply replace b2c.finished.watch with b2c.finishing.watch, because b2c.finishing may occur too early with respect to tc.free. Figure 6.6 depicts the use of resource C during a trip from A to D. Consider the occurrence of event a2b.finishing, it will occur when the the brakeway of the
automatically guided vehicle equals the distance from the front the automatically guided vehicle to the beginning of C, which depends upon the velocity of the automatically guided vehicle. Only at relatively low velocities will a2b.finishing occur when the tail of the automatically guided vehicle is indeed beyond A. At higher velocities the automatically guided vehicle will be further from C when a2b.finishing occurs, and PrimeTickets will have been insisted for all resources within the brakeway. All these brakeway-Tickets must be passed to cToD for freeing. If the velocity remains high then most of these Tickets must extracted from cToD upon termination of cToD.execute as exit-Tickets and passed to the next Action for freeing. In general, the number of exit-Tickets that must be passed from one Action to the next depends upon the velocity of the automatically guided vehicle, and such code will be complicated and not intuitive.

**a concurrent programming alternative**

Concurrent programming provides an alternative that avoids using exit-Tickets; an Action is considered a light-weight sub-process which is responsible for freeing any Ticket it may insist. A concurrent definition of BtoC is presented below, its use is depicted in figure 6.6. This definition also avoids excessive use of Threads by not extending Action1B and providing its own implementation of the Action interface. During construction the required PrimeTicket and Move are created in lines 03-04, both will be downloaded to the Actor together with this Action after creation. The created Move is a BtoC, its definition is presented on page 87 in section 4.2. Upon assimilation the local reference to the Actor and the agv-MFDriver are set in lines 07-08. Upon activation via execute tC is insisted in line 11 and the agv-MFDriver is requested
to begin performing b2c in line 12. The invocation of execute is blocked in
line 13 until b2c.finishing occurs, which is when the automatically guided
vehicle must start decelerating to stop in C. Invocation of all statements in
lines 03-13 are in sequence with respect to each other, and are executed by
the Actor-Thread. The issue of concurrent programming is relevant with re-
spect to the invocation of beyond. This method is invoked upon occurrence of
the event defined by the b2c.beyond Flag-Signal, which is when the tail of the
automatically guided vehicle passes beyond C. At this time the Actor-Thread
will have terminated execute and activated the next Action. The beyond will
be invoked by a Thread originating in the agv-MFDriver.

```
00  class BtoC implements Action {
01      Ticket tC; Move b2c; MFDriver agv; Actor actor;
02      public BtoC(BCScene bcScene) {
03          tC = bcScene.sC.createPrimeticket(1);
04          b2c = new B2C(this,bcScene.locale);
05      }
06      public void assimilate(MFDriver agv,Actor actor) {
07          this.agv = mfd;
08          this.actor = actor;
09      }
10      public void execute() {
11          tC.insist();
12          agv.bgin(b2c);
13          b2c.finishing.watch();
14      }
15      public void beyond() {
16          tC.free();
17      }
18      ...
```

Not all methods defined by the Action interface are implemented above, for
reasons of clarity these implementations are shown below.

```
20      public Ticket getAccessTicket(int i) { if (i==0) { return tC; } }
20      public Ticket getSign(int i) {
21          if (i==0) { return b2c.finishing; }
22          if (i==1) { return b2c.started;  }
23          if (i==2) { return b2c.finished;  }
24          if (i==3) { return b2c.beyond;  }
25          return null;
26      }
```
public Ticket getExitTicket(int i) { return null; }
public Ticket execute(Ticket[] entryTickets) { this.execute() }
public void prepare() {}
public void run() {
    throw new Exception("Action BtoC: run not implemented");
}

In general, a Move will be defined together with the Action that will generate and govern it, by the same traffic control engineer, because the two are so interdependent.

the ‘size’ of an Action

Whenever developing a layout and inferring shared resources, it may occur that a shared resource inferred by the traffic control engineer is smaller then the automatically guided vehicle itself. Suppose an Action is defined for use of a single resource, and the Action implements the behavior described above: upon activation such an Action will insist upon a PrimeTicket providing access to the resource before instructing the MfDriver to use the resources, and it will free the PrimeTicket after the machine no longer requires the resource. This behavior imposes no constraint upon the size of the resource itself. For an automatically guided vehicle, activation of a ‘small’ Action allows it to progress forward only a short distance, from the current resource to the next small resource.

An Action requiring only one resource and implementing the behavior described above is called a basic-Action. In section 9.2 it is discussed how a software tool may generate basic-Actions automatically from a layout if it knows the automatically guided vehicle’s volume.

dealing with transmission latencies

The problem of transmission latencies is introduced on page 68 in section 3.5. Recall the example of automatically guided vehicles traveling through a single lane tunnel. Suppose the tunnel has a capacity of 10 and that there is a single capacity intersection at the end of the tunnel. Consider the following script.
6.2. **BASIC TRAFFIC CONTROL**

```
02    tunnel.execute();
     ...
07    tIntersection.insist();
     ...
09    tIntersection.free();
```

Suppose two automatically guided vehicles are in the tunnel and that both their *Actors* are executing the script above. *Actor-1* has invoked `tunnel.execute` before *Actor-2*, and *agv-1* has indeed entered the tunnel before *agv-2*. Because the tunnel is single lane the automatically guided vehicles can not overtake each other in the tunnel, and the invocation of `tunnel.execute` by *Actor-1* should terminate before that of *Actor-2*. However, an extreme transmission latency may cause `tunnel.execute` by *Actor-2* to terminate first after which it will invoke `tIntersection.insist` first. As a result a deadlock will occur: *Actor-1* will not instruct its automatically guided vehicle to exit the tunnel because it can not acquire permission to use the intersection, and *Actor-2* will never invoke `tIntersection.free` because its automatically guided vehicle can not exit the tunnel. Such a deadlock may be avoided using an *OrderQueue* as follows.

```
01    tunnelQueue.push(agv);
02    tunnel.execute();
03    Transaction txn = AgileSpace.createTransaction();
04    tunnelQueue.pull(agv,txn);
05    tIntersection.reserve();
06    intersectionQueue.push(agv,txn);
07    tIntersection.insist();
08    ...
09    tIntersection.free();
```

Suppose that a manually operated road-block guarantees that the order of objects representing automatically guided vehicles in the `tunnelQueue` do indeed correspond with the order of automatically guided vehicles in the tunnel. If so, then *Actor-2* can not pull its `agv` from the `tunnelQueue` before *Actor-1*. Because *Actor-1* pulls its `agv` from the `tunnelQueue` under a transaction *Actor-2* can not do so until after *Actor-1* pushed its `agv` in `intersectionQueue`. This implies *Actor-1* will invoke `tIntersection.reserve` first, regardless of any transmission latency. Transmission latencies may allow *Actor-2* to ‘overtake’ *Actor-1* after `intersectionQueue.push` and invoke `tIntersection.insist` first, but this is not a problem because *Actor-1* has reserved `tIntersection` first. Note that lines 03-06 are non-blocking and that little time will be lost during their execution.
6.3 Monitoring Action Events

Upon activation an Action starts generating Action events. In this section it is discussed how such events must be defined, monitored, and propagated to an interested Agent.

defining events

In section 6.1 the four standard Action-events are introduced, and these events may be monitored as Signs. Suppose a traffic control engineer wishes to define a non-standard event halfwayThere upon Action AtoE. Consider the following definition.

```java
class AtoE extends ActionIB {
    ActionIB aToC;
    ActionIB cToE;
    public Sign halfwayThere = new Sign();
    private void script() { ... }
}
```

The super-Action may monitor all five events generated by an AtoC as follows.

```java
AtoE aToE = abcdeScene.createAction("AtoE");
aToE.run();
aToE.started .watch();
aToE.halfWayThere .watch(); // agv is in C now
aToE.finishing .watch();
aToE.finished .watch();
aToE.beyond .watch();
```

The traffic control engineer must also define when the events will occur. Consider the AtoE-script below, in lines 01 through 04 the four standard events are associated with events generated by the sub-Actions. The custom halfwayThere event is associated with aToC.finished when the automatically guided vehicle is indeed half-way between A and E. Note that the halfwayThere event is generated after cToE.execute is invoked, and regardless of the speed with which the automatically guided vehicle travels through C.
class AtoE extends ActionIB {
    ...
    private void script() {
    01    aToC.started = this.started;
    02    cToE.finishing = this.finishing;
    03    cToE.finished = this.finished;
    04    cToE.beyond = this.beyond;
    05    aToC.finished = this.halfWayThere;
    06    aToC.execute();
    07    cToE.execute();
    } }

propagating events up the Action-tree

In the example above Action AtoE propagates events to its super-Action without monitoring them itself. However, it may be true that the AtoE is also interested in events and in order to undertake some activity when a particular event occurs. Consider the following implementation for AtoE.

class AtoE extends ActionIB {
    ...
    private void script() {
        aToC.started = this.started
        aToC.finishing = new Sign();
        cToE.finishing = this.finishing;
        cToE.finished = this.finished;
        cToE.beyond = this.beyond;
        aToC.finished = this.halfWayThere;
        aToC.run();
        aToC.started.watch();
        aToC_started();
        aToC.finishing.watch();
        aToC_finishing();
        cToE.execute();
    }
    void aToC_started() { ... }
    void aToC_finishing() { ... }
}

Upon occurrence of aToC_started method aToC_started is invoked, and upon occurrence of aToC_finishing method aToC_finishing is invoked. The AtoE
script is straight forward but implies a particular order in which events can
be handled: aToC.started must be handled before aToE.finishing. However, aToE.finishing may occur before aToC.started if the automatically guided vehicle is traveling fast enough. As a result, cToE.execute may be invoked later than intended. This issue can be avoided by using Signals. Consider the following alternative definition for AtoE.

```java
public class AtoE extends ActionIB {
    public void script() {
        aToC.started = new Signal(aToC, this, "aToC_started" );
        aToC.finishing = new Signal(aToC, this, "aToC_finishing");
        cToE.finishing = this.finished;
        cToE.beyond = this.beyond;
        aToC.finished = this.halfWayThere;
        aToC.execute();
        cToE.execute();
    }
    public void aToC_started() { this.started.send(); ... }
    public void aToC_finishing() { ... }
}
```

This last alternative implementation of AtoE defines an Action that behaves in a manner identical to that an AtoE defined using the original AtoE-script discussed at the beginning of this section. However, the event-handling methods could also contain code to perform any other activity deemed important by the traffic control engineer such as insisting and freeing Tickets or sending Letters to other Agents.

### 6.4 Exceptions

The first versions of the mini-agv allowed wheel-servo settings that exceeded mechanical limits. When turning a sharp corner, the wheel-servos would attempt to pull the wheels into a mechanically impossible angle, and would overheat. If the mini-agv would stop during such a sharp corner, the wheel-servos would start burning and melt into a fixed angle. After the mini-agv would continue on its way, it would not be able to steer and travel in the wrong direction. Due to its trace-following intelligence, it would continuously try to correct this situation, but without success. As a result, the mini-agv would veer off its prescribed FuTrace and display erratic behavior, and could collide with other mini-agv in its vicinity. It would be preferred it the traffic
control system could detect the problem, and react to it, automatically. This has not been implemented, but a possible approach is discussed below. First, the current non-automated recovery procedure is discussed.

The traffic monitor is the person who starts the DemoScene from the command line, and introduces the mini-agvs into the transport system. Traffic monitor would see the mini-agv experiencing problems, possibly after being alerted by the smell of the burning wheel-servos, and remove the burning mini-agv from the infrastructure guarded by DemoScene. After the mini-agv is physically removed, its Actor and MFDriver must be removed from the traffic control system residing in the virtual world. To do this, the mini-agv was equipped with a custom dispose command, which can be called via the telnet-console that is also used to start the mini-agv software. This command is results in the following activities.

- the agv-Actor will invoke finalize upon its job-Action, which had been downloaded from the DemoScene, and is implemented to free all Tickets. As a result, all infrastructure resources would be available to other agv-Actors once again, the the other automatically guided vehicles would continue on their way.

- the agv-Actor informs the LoGoS Agent that passed it a job, that it is no longer functioning, and it retracts its advertisement of its services within AgileSpace.

- the mini-agv’s MFDriver retracts its advertisement of its services within AgileSpace, after which the local positioning system, and the local 3d-visualization system, would stop providing their services to the mini-agv’s MFDriver.

- the java runtime environment upon the mini-agv shuts down.

It would be preferred if the traffic control system could detect a failed mini-agv, and react to such a failure, automatically. To do this, the DemoScene-Moves must generate an event whenever the FuTrace following deviation exceeds a particular value. Such an event should be modelled as a Flag-Signal, because the event will occur at an unpredictable moment. Suppose the Flag-Signal invokes deviationTooLarge, then DemoScene-Actions must implement this method, and inform the Scene upon invocation. The Scene is aware of all Actions that have been downloaded by Actors. The Scene and its Actions could be programmed in such a way that the Scene could instruct downloaded and active Actions to invoke mfd.adaptspeed(0) after which the automatically guided vehicle they govern will stop immediately, hopefully before a collision occurs.
After all automatically guided vehicles have stopped, the traffic monitor must be requested to do something about the situation, possibly by sounding an alarm. After the traffic monitor removes the mini-agv and requests the mini-agv to dispose itself, as described above, the Scene could instruct Actions to invoke \texttt{mfd.adaptspeed(1)}, after which all mini-agvs continue on their way.

The crossover traffic control system is distributed and communication links may fail. Within mini-world such failures occurred due to accidental removal of a network wire from a computer or a communications hub. Upon occurrence of such a failure, a \texttt{RemoteException} will be generated during a remote invocation of methods over such a failed communications link. There is no general solution to this problem, which is inherent with respect to distributed programming, and traffic control engineers must decide how to handle \texttt{RemoteExceptions} whenever they design custom interaction between \texttt{Agents}. Regarding interaction between \texttt{Actors} and \texttt{Scenes}, or rather between \texttt{PrimeTickets} and \texttt{Semaphores}, these software components simply keep re-trying a remote method invocation until the communications link is restored. This implies invocation of \texttt{insist} on a \texttt{PrimeTicket} may block even after capacity has been assigned by its \texttt{Semaphore}, and the \texttt{Action-script} will stall until the communications link is restored. Such fail-safe behavior is a requirement imposed by traffic control.
Chapter 7

The Crossover Terminal

Figure 7.1: layout of the crossover terminal

The crossover container terminal is an envisioned sea-terminal that must be able to service *jumbo container ships* within 24 hours. Regarding transport of containers to and from a jumbo container ship, the crossover transport system is specifically designed to meet this requirement. Implementation of the CrossoverScene, the TrAcEs traffic control Agent of the crossover transport system, is the subject of this chapter. In section 7.1 the layout the of crossover transport system is presented. In section 7.2 it is shown how CrossoverScene is defined, and how an *Actor* interacts with CrossoverAgy, which is a LoGoS *Agent* responsible for logistic control of an *Actor* as it circulates within the crossover transport system. Section 7.3 introduces the Circulate *Action*, and its sub-*Actions*, and shows how traffic intelligence, required to circulate within the crossover transport system, is modelled. Section 7.4 shows how simulation of *Agents* external to the TrAcEs sub-system allows the CrossoverScene to be validated.

149
7.1 The Crossover Layout

The lanes automatically guided vehicles follow greatly effect the performance of an automated transport system. Figure 7.1 depicts the crossover layout, automatically guided vehicles crossover directly from the stack to the quay. The layout attempts to minimize the interactions between automatically guided vehicles, the intuition is that this will result in smaller standard deviations for the performance mean time of transport jobs; this has been demonstrated in simulations [44]. The crossover layout is defined using FuTraces within XYASpace that an automatically guided vehicle may follow.

travelling from a stack area to a quay area

Container stacking lanes are used to store containers. Figure 2.3 depicts 12 automated container stacking lanes positioned at the land-side of the crossover transport system. Figure 7.1 depicts 12 stack areas, with four stack park areas each, that the crossover layout provides for every container stacking lane. The automatic container stacking crane may exchange containers with automatically guided vehicles waiting upon a stack park area. After a container exchange, an automatically guided vehicle must travel from the stack area to one of the six quay cranes servicing a jumbo container ship. Each quay crane is equipped with an agv-turntable; it is used to orientate an automatically guided vehicle before a container exchange. In figure 7.1 only the agv-turntables are depicted. Before an automatically guided vehicle travels onto an agv-turntable, it will wait for its turn in a quay area, which is composed of 5 quay parking areas. In figure 7.1, 11 quay areas are depicted just north of the quay crane trench. If the destination quay area is one of the four directly to the south of the stack area, the automatically guided
7.1. THE CROSSOVER LAYOUT

Figure 7.3: FuTraces for travelling from a quay area to an agv-turntable

vehicle travels to the destination quay area by crossing over directly. If the destination quay area is to the west, the automatically guided vehicle travels westward along the northern crossover lane, and crosses over to the south when the destination quay area can be reached. If the destination quay area is to the east, the automatically guided vehicle crosses over to the southern crossover lane and travels eastward to the destination quay area. Traffic crossing over has higher priority than traffic driving on a crossover lane. For this reason a crossover lane is subdivided into parking areas and transit areas. The location of a parking area is such that a parked automatically guided vehicle does not interfere with an automatically guided vehicle crossing over in southern direction via a transit area. Figure 7.2 depicts the nine types of FuTraces used to define the crossover layout.

travelling from a quay area to an agv-turntable

An agv-turntable is fixed between the lower legs of a quay crane, it moves together its quay crane along the quay. The turntable is level with the quay; the quay crane trench provides space for the turning mechanism. An automatically guided vehicle will enter an agv-turntable at an angle varying from north-west to north-east. The layout for traveling from a stack park area to an agv-turntable is defined by one type of FuTrace only, all FuTraces depicted in figure 7.3 are instances of this single type. Upon arrival an agv-turntable will rotate the automatically guided vehicle in a desired eastern or western orientation before a container is exchanged.

travelling from an agv-turntable to a stack area

After a container exchange with the a quay crane, the automatically guided vehicle will be rotated to a southern orientation before it departs. Assuming it departs in western direction, it will first travel to the south-western corner via the southern lanes, then via the western lanes to the north-western corner, and on to the northern lanes. figure 2.3 depicts how the two northern lanes are positioned between the 12 container stack lanes and the 12 stack
areas. An automatically guided vehicle will enter a stack parking area from a northern lane, and wait for a container exchange with the automatic stacking crane. The automatic stacking crane stores an retrieves containers in the container stack lane, it does not interfere with traffic upon the northern lanes as it passes by overhead.

7.2 The Crossover Agents

The crossover transport system is controlled by several cooperating Agents. Traffic is controlled by a CrossoverScene TrAcEs Agent, it provides Circulate Actions to Actors. In this section it is shown how the CrossoverScene is defined. Upon arrival at the crossover transport system, an Actor is requested to perform a CrossoverScene-Circulate job by a CrossoverAgy LoGoS Agent, which is responsible for logistic control of an Actor as it circulates upon the crossover terminal. This means a CrossoverAgy must inform an Actor when to start traveling to its destination stack area or quay area, and when to stop circulating. This interaction is defined within Circulate and its sub-Actions. Logistic control is not a subject of this thesis, but in this section it is shown how a CrossoverAgy should behave. In section 7.4 it is shown how this behavior is implemented, in a trivial manner, within the context of a simulation.

the CrossoverScene traffic control Agent

The layout of the crossover terminal defines many intersections and parking areas, 7.4 depicts how these have been interpreted as shared resources that are guarded by BasicSemaphores modelling a single unit of capacity. The most challenging aspect of the crossover transport system is controlling automatically guided vehicles as they cross over from the stack areas to quay cranes. The relevant Semaphore declarations with respect to crossing over are presented below. The complete definition of CrossoverScene is presented is appendix A.

```java
public class CrossoverScene implements Scene {
    XYATransform locale;
    int numberOfStacks;
    ...
    Semaphore sSPA [] ;
    Semaphore sSPAX [] ;
```
Figure 7.4: Guarded Resources

Semaphore sPCN  
Semaphore sPCC  
Semaphore sPCS  
Semaphore sTCN  
Semaphore sTCC  
Semaphore sTCS  
Semaphore sQPAX  
Semaphore sQPA  
Semaphore sQC  

\[
\text{public CrossoverScene}(\text{XYATransform locale}, \text{int numberOfStacks}, \text{ServiceID serviceID}) \{ \ldots \}
\]

\[
\text{public Action createAction}(\text{String actionClassName}) \{ \ldots \}
\]

CrossoverScene(\text{XYATransform locale},\text{int numberOfStacks},\text{ServiceID serviceID}): constructs this CrossoverScene according to the locale of the crossover
transport system within the environment, the number of container stacking lanes, and the service-identifier. The locale of a CrossoverScene is depicted in figure 7.1. A CrossoverScene registers the Scene-Service it provides with AgileSpace using the specified ServiceID. The number of Semaphores created during construction depends on the size of the crossover terminal, which in turn depends upon the number of stack lanes. For example, the BasicSemaphores guarding quay park areas are created as follows.

```java
for(int i=0;i<numberOfStacks;i++) {
    for(int j=0;j<4;j++) {
        sQPA[i][j] = new BasicSemaphore(1);
    }
}
```

Action createAction(String actionName): return a new Circulate if actionName equals "Circulate". A Circulate contains the traffic intelligence required to circulate within the crossover transport system. It definition, including its sub-scripts and sub-Actions, is the subject of section 7.3.

the CrossoverAgv logistic control Agent

Upon arrival at the northern side of the crossover transport system, the agv-Actor will be assigned a job by a CrossoverAgv LoGoS Agent. A CrossoverAgv is aware of the ServiceID of the local CrossoverScene, and is aware of the "Circulate" Action. It will use these values to define a Job for the newly arrived agv-Actor. How the CrossoverAgv is aware of these values, and of the newly arrived agv-Actor, is beyond the scope of this thesis. However, it is shown how this is achieved within the context of a simulation in section 7.4. After activation within the context of an Actor, a Circulate must interact with the LoGoS Agent that instructed the Actor to activate it, via the following interface.

```java
interface CrossoverAgv extends Service {
    boolean toCirculate();
    int getStackAreaID(int side);
    int getQuayAreaID();
    int getQuayCraneID(int quayParkAreaID);
    XYASpace getQuayCraneLocale()
    void arrivedAtQuayCrane();
}
```
toCirculate: returns true iff the automatically guided vehicle must travel from the stack area to a quay crane and back, false if the automatically guided vehicle must exit from the crossover terminal.

getStackAreaID(int side): returns destination stack area's integer-identifier. The side-parameter indicates if the automatically guided vehicle is on the western or the eastern side of the crossover transport system.

getQuayAreaID: returns destination quay area's integer-identifier. The invocation will block until the automatically guided vehicle may depart, in this way a CrossoverAgv can time the arrival of containers at quay cranes.

getQuayCraneID(int quayParkAreaID): returns the integer-identifier of the destination quay crane.

getQuayCraneLocale: returns the locale of the quay crane at this time, the local will not change until arrivedAtQuayCrane is invoked.

arrivedAtQuayCrane: informs the quay crane that the automatically guided vehicle is on the agv-turntable of the quay crane of which the locale was previously returned.

A CrossoverAgv will pass this interface along when it requests the agv-Actor to perform the job. An activated Circulate may acquire a reference to CrossoverAgv via this.actor.getService. How the CrossoverAgv Agent knows which values to return is an issue beyond the subject of this thesis. However, in section 7.4 it is shown how CrossoverAgvSimulator computes values that are useful in order to test the crossover transport system within the context of a simulation.
7.3 The Crossover Actions

Circulate is the only Action provided by CrossoverScene to Actors. It is composed of several sub-Actions and sub-scripts and is defined, in part, as follows. The complete definition of Circulate is presented in appendix A.

```java
public class Circulate extends ActionIB {

    int stackAreaID, stackParkAreaID, quayAreaID;

    // subActions
    stackAreaToWestToQuayArea[][][]
    stackAreaToSouthToQuayArea[][][]
    stackAreaToEastToQuayArea[][][]
    ...
    public Circulate(CrossoverScene crossoverScene) { ... }
    private void script() { ... }
    private boolean toCirculate() { ... }
}
```

- **int stackAreaID**: the index of the stack area where the automatically guided vehicle must, or has, exchanged a container with an automatic stacking crane.

- **int stackParkAreaID**: the index of the stack park area where the automatically guided vehicle must, or has, exchanged a container with an automatic stacking crane.

- **int quayAreaID**: the index of the quay area where the automatically guided vehicle must wait, or has, waited for its turn to travel to the agv-turntable.

- **stackAreaToWestToQuayArea[stackAreaID][stackParkAreaID][quayAreaID]**: array of sub-Actions for traveling from a stack park area within a stack area, to a quay situated west with respect to the stack area. Upon activation, the automatically guided vehicle will travel westward along the northern crossover lane, and cross over when the destination quay area can be reached.

- **stackAreaToSouthToQuayArea[stackAreaID][stackParkAreaID][quayAreaID]**: array of sub-Actions for traveling from a stack park area within a stack area,
7.3. **THE CROSSOVER ACTIONS**

to a quay situated south with respect to the stack area. Upon activation, the automatically guided vehicle will travel to the destination quay by crossing over directly.

```
stackAreaToEastToQuayArea[stackAreaID][stackParkAreaID][quayAreaID]: array of sub-Actions for traveling from a stack park area within a stack area, to a quay situated west with respect to the stack area. Upon activation, the automatically guided vehicle will cross over to the southern crossover lane, and travels eastward until the destination quay area can be reached.
```

**Circulate(CrossoverScene crossoverScene):** constructs this Action and its sub-Actions. The exact number of sub-Actions created depends on the size of the crossover terminal, in the same way the exact number of Semaphores created depends on the size of the crossover terminal. For example, the sub-Actions required for traveling from a stack area to a quay area via the northern crossover lane are created as follows.

```
for (stackAreaID=0; stackAreaID<12; stackAreaID++) {
  for (stackParkAreaID=0; stackParkAreaID<4; stackParkAreaID++) {
    for (quayAreaID=0; quayAreaID<12-1; quayAreaID++) {
      stackAreaToWestToQuayArea
        [stackAreaID][stackParkAreaID][quayAreaID] =
        new StackAreaToWestToQuayArea(
            cS, stackAreaID, stackParkAreaID, quayAreaID);
    }
  }
}
```

The construction of a StackAreaToWestToQuayArea is discussed further on page 159.

**assimilate:** assimilates all the sub-Actions.

**script:** instructs an Actor to circulate upon the crossover transport system until further notice. In line 01 a sub-script is activated which instructs the automatically guided vehicle to enter the crossover transport system and to travel to a stack park area. In line 06 a sub-script is activated which instructs the automatically guided vehicle to exit the crossover transport system from a stack park area. In lines 02 to 05 the automatically guided vehicle is instructed to circulate until further notice.
CHAPTER 7. THE CROSSOVER TERMINAL

```plaintext
01 enterToQuayArea(...) 
02 while ( toCirculate() ) { 
03 stackAreaToQuayCrane(); 
04 quayCraneToStackArea(); 
05 } 
06 quayAreaToExit() 
```

The sub-scripts will interact with the CrossoverAgyv in order the know which stack area to travel to, which quay crane to travel to, and when to stop circulating. How this interaction is achieved within toCirculate is shown below. Upon receiving the desired information sub-scripts will activate the appropriate sub-Actions. How this is achieved for stackAreaToQuayCrane is shown in the rest of this section.

```plaintext
boolean toCirculate: returns true if the Actor must continue circulating, false otherwise. This method requests the CrossoverAgyv which value it must return, as follows.

crossoverAgyv = (CrossoverAgyv)actor.getService();
return crossoverAgyv.toCirculate();
```

**Action Circulate Sub-script stackAreaToQuayCrane**

An automatically guided vehicle will not travel directly to a quay crane, it will first wait for its turn in a quay area. In order to travel to the destination quay area, the appropriate sub-Action of Circulate must be activated. Before invocation of stackAreaToQuayCrane, the automatically guided vehicle has exchanged a container in a particular stack park area. The index of the stack area is stored in stackAreaID, and the index of the stack’s park area is stored in stackParkAreaID. Upon invocation of stackAreaToQuayCrane, the CrossoverAgyv is requested for the index of the destination quay area by executing crossoverAgyv.getQuayAreaID. In the destination quay area the automatically guided vehicle will wait for its turn to travel to the agv-turntable of the destination quay crane. The appropriate sub-Action is selected and activated as follows.
Figure 7.5: stackAreaToWestToQuayArea[11][1][1]

```java
public void stackAreaToQuayCrane() {
    crossoverAvg = (CrossoverAvg)actor.getService();
    int quayAreaID = crossoverAvg.getQuayAreaID();
    if (quayAreaID>=stackAreaID+2) {
        stackAreaToEastToQuayArea[stackAreaID]
            [stackParkAreaID]
            [quayAreaID].execute();
    }
    else if (quayAreaID<=stackAreaID-3) {
        stackAreaToWestToQuayArea[stackAreaID]
            [stackParkAreaID]
            [quayAreaID].execute();
    }
    else {
        stackAreaToSouthToQuayArea[stackAreaID]
            [stackParkAreaID]
            [quayAreaID].execute();
    }
}
```

Action StackAreaToWestToQuayArea

Figure 7.5 depicts the infrastructure required to travel from stack area 11 - stack park area 1, to quay area 1 - quay park area 4. This route is modelled by sub-Action StackAreaToWest, which is also composed of sub-Actions. The issue of interest is the use of a SelectTicket in order to determine the destination quay park area. The complete definition of StackAreaToWestToQuayArea is presented in appendix A, the part relevant to the issue of interest is presented below.
public class StackAreaToWestToQuayArea extends ActionIB {

    private StackAreaToWest stackAreaToWest;
    private CrossoverAreaToWest[] crossoverAreaToWest;
    private CrossoverAreaToQuayAreaWest crossoverAreaToQuayAreaWest;

    public StackAreaToWestToQuayArea(CrossoverScene cS,
                                        int stackAreaID,
                                        int stackParkAreaID,
                                        int quayAreaID) { ... }

    public void assimilate(Actor actor) { ... }
    protected void script() { ... }
}

stackAreaToWest: sub-Action for traveling from a stack park area within a
stack area, to the first parking area upon the northern crossover lane.

crossoverAreaToWest[stackAreaID]: array of sub-Actions for traveling from a
parking area upon the northern crossover lane to the next, in western
direction.

crossoverAreaToQuayAreaWest: Sub-Action for crossing over from a parking area
upon the northern crossover lane to a quay park area.

public StackAreaToWestToQuayArea: constructs this Action and its sub-Actions.
The exact number of sub-Actions created depends on the distance be-
tween the origin stack area and the destination quay area. The sub-
Actions are created as follows.

    stackAreaToWest =
        new StackAreaToWest(cS,stackAreaID,stackParkAreaID);
    for (int nr=(stackAreaID-2); nr>=(quayAreaID+2); nr--) { 
        crossoverAreaToWest[nr] = new CrossoverAreaToWest(cS,nr);
    }
    crossoverAreaToQuayAreaWest =
        new CrossoverAreaToQuayAreaWest(cS,quayAreaID);

assimilate: assimilates all the sub-Actions, as follows.

    stackAreaToWest.assimilate(actor);
    for (int nr=(stackAreaID-2); nr>=(quayAreaID+2); nr--) {
        crossoverAreaToWest[nr].assimilate(actor);
    }
    crossoverAreaToQuayAreaWest.assimilate(actor);
script: selects a quay park area within the destination quay area and activates the appropriate Actions, as follows. In lines 01-05 sub-Action crossoverAreaToQuayAreaWest is requested to pass references to access-Tickets, these are the Tickets providing access to the five quay park areas. In lines 06-08 a SelectTicket is used to select the destination quay park area. In line 09 stackAreaToWest is activated after which the automatically guided vehicle will travel to the first parking area upon the northern crossover lane. In lines 10-12 a sequence of crossoverAreaToWest is activated after which the automatically guided vehicle will travel along the northern crossover lane until it can reach the destination quay area. In line 13 crossoverAreaToQuayAreaWest is activated after which the automatically guided vehicle will travel from the final parking area upon the northern crossover lane to its destination quay park area.

```java
01  tQPA0 = crossoverAreaToQuayAreaWest.getAccessTicket(0);
02  tQPA1 = crossoverAreaToQuayAreaWest.getAccessTicket(1);
03  tQPA2 = crossoverAreaToQuayAreaWest.getAccessTicket(2);
04  tQPA3 = crossoverAreaToQuayAreaWest.getAccessTicket(3);
05  tQPA4 = crossoverAreaToQuayAreaWest.getAccessTicket(4);
06  SelectTicket quayParkAreaST = new SelectTicket(
06      tQPA0,tQPA1,tQPA2,tQPA3,tQPA4
06  );
07  quayParkAreaST.insist();
08  quayParkAreaID = quayParkAreaST.getSelectedIndex();
09  stackAreaToWest.execute();
10  for (int nr=(stackAreaID-2); nr>=(quayAreaID+2); nr--) {
11     crossoverAreaToWest[nr].execute();
12  }
13  crossoverAreaToQuayAreaWest.execute(quayParkAreaID);
```

Note that quayParkAreaST is not freed within script, this is because the selected Ticket will be freed by sub-Action crossoverAreaToQuayAreaWest when the automatically guided vehicle has departed to the agv-turntable. This is achieved by associating a beyond-event with the contained Move, in a manner similar to CrossoverAreaToWest, which is discussed below, and freeing the selected Ticket upon occurrence of that event.

Action StackAreaToWest

Figure 7.6 depicts the infrastructure used by automatically guided vehicles as they are governed by a StackAreaToWest. The crossover layout has been
designed in such a way that automatically guided vehicles waiting in parking areas will not interfere with traffic crossing over. It seems best that an automatically guided vehicle should travel from the stack park area, to the destination park area upon the northern crossover lane, in a single haul. To do this, all resources must be acquired before the MFDriver is instructed to perform the contained Move. The issue of interest is that it requires a set of resources also required by other Actions, and deadlocks must be avoided. The complete definition of StackAreaToWest is presented in appendix A, the part relevant to the issue of interest is presented below.

```java
public class StackAreaToWest extends ActionIB {
    PrimeTicket[] tSPAX;
    PrimeTicket tTCN;
    PrimeTicket tPCN;
    StackArea2West stackArea2West

    public StackAreaToWest(
            CrossoverScene cS, int stackAreaID, int stackParkAreaID) {
        ...
    }

    protected void script() {... }
    public void beyondStackArea() {... }
    public void finished() {... }
    public void beyond() {... }
}
```

tSPAX: array of PrimeTickets created via Semaphores sSPAX[stackAreaID][0-3] and sSPAX[stackAreaID-1][2-3]. These Semaphores guard stack exit areas, which are used when the automatically guided vehicle turns westward as it departs from the stack area.

tTCN: a PrimeTicket created via Semaphore sTCN[stackAreaID-1]. This Semaphore guards the transit area upon the northern crossover lane, it is used as
the automatically guided vehicle crabs to the destination park area upon the northern crossover lane.

tPCN: a PrimeTicket created via Semaphore sPCN[stackAreaID-2]. This Semaphore guards the destination park area upon the northern crossover lane.

stackArea2West: the Move this Action governs, it contains a FuTrace depicted in figure 7.6. This Move is not discussed in detail, but its definition is presented in appendix A. A StackArea2West defines four Flags. A Flag-Sign is raised when the automatically guided vehicle must start decelerating in order to stop in the destination park area. A Flag-Signal is raised when the automatically guided vehicle merges on the northern crossover lane, method beyondStackArea will be invoked. A Flag-Signal is raised when the automatically guided vehicle arrives in the destination park area, method finished will be invoked. A Flag-Signal is raised when the tail of the automatically guided vehicle travels beyond the destination park area, method beyond will be invoked.

StackAreaToWest: constructs the Action. The parameters are needed to create the PrimeTickets providing access to the required resources, and for creating the appropriate Move. Sign-0 is set to be stackArea2West.finishing, this implies the invocation of execute will terminate when the automatically guided vehicle must start decelerating in order to stop in the destination park area.

script: models the traffic intelligence contained within this Action. In line 01 the destination park area upon the northern crossover lane is acquired. In line 03 all the Tickets providing access to the exits of stack park areas are insisted via a CollectTicket, why this is so is discussed below. In line 04 the transit area upon the norther crossover lane required to travel to the destination park area is acquired. In line 05 the automatically guided vehicle is requested to travel from the stack park area to the park area upon the northern crossover lane.

01 tPCN.insist();
02 collectTicket = new CollectTicket(tSPAX);
03 collectTicket.insist();
04 tTCN.insist();
05 actor.getMFDriver().begin(stackArea2West);
public beyondStackArea: frees the Tickets providing access to the stack exit areas, as follows.

    for (i=0;i<tStackExit.length;i++) { tSPAX[i].free(); }

public finished: invokes tTCN.free.

public beyond: invokes tPCN.free.

Within script, a CollectTicket is used to insist upon PrimeTickets providing access to the exit of the stack park areas the automatically guided vehicle must use. The reason for this is a possible deadlock with other Actions that may also require these resources at the same time. Figure 7.7 depicts the infrastructure used by automatically guided vehicles as they are governed by a StackAreaToWest. A StackAreaToWest and a StackAreaToEast both use stack exit areas. Figure 7.8 depicts the resources required by an activated stackAreaToWest[i][j] and stackAreaToEast[i−1][k]. An activated stackAreaToWest[i][j] and stackAreaToEast[i−1][k] will require the same stack exit resources. Traffic control engineers responsible for designing these Actions could avoid such a deadlock by agreeing upon an order in which these resources must be claimed. However, it would be preferred if the traffic control engineers could do their work independently. Assuming a traffic

Figure 7.8: resources used by a StackAreaToWest and a StackAreaToEast
control engineer can infer from the intended use of the layout that there is a conflict regarding the stack exit areas, then the traffic control engineer should insist upon these conflicting resources in an atomic manner. This insight is modelled in the script of StackAreaToWest above. The same approach is used in the definition of StackAreaToEast, its definition is presented in appendix A.

Action CrossoverAreaToWest

Figure 7.9 depicts the infrastructure used by automatically guided vehicles as they are governed by a CrossoverAreaToWest. For the same reasons as for StackAreaToWest, it seems best that the automatically guided vehicle should follow its contained FuTrace in a single haul. The complete definition of CrossoverToWest, together within Move it governs, is presented below.

class CrossoverAreaToWest extends ActionIB {
    Ticket[] tTCN;
    Ticket[] tPCN;
    CrossoverArea2West crossoverArea2West;

    public CrossoverAreaToWest(CrossoverScene cS ,
                               int stackAreaID ) { .. }

    public void script() { ... }
    public void finished() { ... }
    public void beyond() { ... }
}

tTCN: a PrimeTicket created via a sTCN[stackAreaID]. This Semaphore guards the transit area between the origin park area and the destination park area, upon the northern crossover lane.

tPCN: a PrimeTicket created via a sPCN[stackAreaID-1]. This Semaphore guards the destination park area upon the northern crossover lane.

crossoverArea2West: the Move this Action governs, it contains a FuTrace depicted in figure 7.9. The definition of this Move is presented below.
CrossoverAreaToWest: constructs this Action, as follows. First the PrimeTickets are created, then the Move is created in line 02. The Move is passed the locale of the crossover transport system, and the value of stackAreaID, so that it can compute the locale of its contained FuTrace within the environment. In line 03, Sign-0 is set to be the finishing event of crossoverArea2West, this implies invocation of execute upon this Action will terminate when the automatically guided vehicle must start decelerating for the destination park area.

```
00  tTCN[stackAreaID ] =
00     cS.sTCN[stackAreaID ].createPrimeTicket(1);
01  tPCN[stackAreaID-1] =
01     cS.sPCN[stackAreaID-1].createPrimeTicket(1);
02  crossoverArea2West =
02     new CrossoverArea2West(cS,stackAreaID,this);
03  this.setSign(0,crossoverArea2West.finishing);
```

Statement line 02 is defined as if the CrossoverScene were implemented using the FoRcEs workbench version 1.0. However, the CrossoverScene was implemented using a prototype of the FoRcEs workbench. This means statement line 02 is pseudo-code, and reflects the definition desired by the author. The actual definition of CrossoverAreaToWest is presented in appendix A, there the issue regarding FoRcEs workbench versions with respect to statement line 02 is also discussed.

script: insists upon both Tickets, before instructing the MFDriver to perform the Move. In this way the contained FuTrace will be followed in a single haul.

```
tTCN[stackAreaID].insist();
tPCN[stackAreaID-1].insist();
agvMfd.begin(crossoverArea2West);
```

finished: invokes tTCN[stackAreaID].free(). Move CrossoverArea2West defines a Flag-Signal that will invoke this method when the automatically guided vehicle has arrived in the destination park area, and the tail of the automatically guided vehicle has passed beyond TCN[stackAreaID].

beyond: invokes tPCN[stackAreaID-1].free(). Move CrossoverArea2West defines a Flag-Signal that will invoke this method when the tail of the automatically guided vehicle has passed beyond TPN[stackAreaID-1].
Within script tPCN[stackAreaID-1] is insisted before tTCN[stackAreaID]. Suppose permissions were acquired the other way around and PCN[stackAreaID-1] is occupied by another automatically guided vehicle that has experienced mechanical failure. If so, permission will be acquired to use TCN[stackAreaID], but not for PCN[stackAreaID-1]: this Action will block any Actor intending to travel directly from a stack area in southern direction via TCN[stackAreaID] for a long time without a good reason.

**Move CrossoverArea2West**

A CrossoverArea2West is created and governed by a CrossoverAreaToWest. The contained FuTrace is a Straight defined in XYASpace, from the origin park area upon the northern crossover lane, to the destination park area on the same lane, in western direction. A CrossoverArea2West also defines Flags that define events that are of interest to the CrossoverAreaToWest that governs it. A pseudo-code definition of CrossoverArea2West is presented below, it is designed as if CrossoverScene-Moves were implemented using the FoRCeS workbench version 1.0. However, CrossoverScene-Moves were implemented using a prototype of the FoRCeS workbench. The actual definition of CrossoverArea2West is presented in appendix A. In the appendix the issue regarding FoRCeS workbench versions with respect to this definition is also discussed.

```java
class CrossoverArea2West extends MoveIB {

    public CrossoverArea2West(
        CrossoverScene cS, int stackAreaID, Action action) { ... }

    Transform computeGlobalLocale(
        CrossoverScene cS, int stackAreaID) { ... }

    FuTrace computeCrossoverArea$West(
        CrossoverScene cS, Transform locale) { ... }
}
```

CrossoverArea2West: constructs this Move, the cLocale is the locale of the crossover transport system within the environment, stackAreaID indicates locale of this Move relative to the locale of the crossover terminal. In line 00 the trace intelligence parameters are set, the same set of parameters is used in the creation of a Move presented in section 4.2. The locale of the contained FuTrace is computed in line 01 according to computeGlobalLocale, which is discussed below. In line 03 the contained FuTrace is created. In lines 04 to 07 three Flags are created that define events of interest to the CrossoverAreaToWest governing this Move. In line
the standard finishing-Flag is created; it is a Flag-Sign that will be raised when the automatically guided vehicle must decelerate in order to stop in PCN[stackAreaID-1]. In line 05 the standard finished-Flag is created; it is a Flag-Sign that will invoke method finished upon the governing CrossoverAreaToWest when the automatically guided vehicle has arrived at the end of the FuTrace. In line 06 the standard beyond-Flag is created; it is a Flag-Sign that will invoke method beyond upon the governing CrossoverAreaToWest when the automatically guided vehicle has travelled its length beyond the end of the FuTrace.

00 super(15,1,0.5,1.5,1.5);
01 Transform globalLocale =
02 computeGlobalLocale(cS,stackAreaID);
03 trace = new Straight(globalLocale,cS.STACK_WIDTH_INCL);
04 double tEnd = agvTrace.getTraceEnd();
05 finishing =
06 new BrakewayFlagSign (trace,tEnd,action);
07 finished =
08 new EvolutionFlagSignal(trace,tEnd,action,"finished");
09 beyond =
10 new EvolutionFlagSignal(trace,tEnd+17.0,action,"beyond");

computeGlobalLocale(Transform crossoverLocale,int stackAreaID): Computes the locale of this CrossoverArea2West relative to the CrossoverTerminal, as follows.

\[ \text{float relativeX} = (\text{stackAreaID} + 1) \times \text{cS.STACK_WIDTH_INCL} + \text{cS.FREE_TERMINAL_LEFT} + \text{cS.FREE_SPACE_AT_SIDES} + \text{cS.WIDTH_DRIVING_LANE} \]

\[ \text{float relativeY} = \text{cS.getStackLaneY()} - \text{cS.FREE_STACK_EXIT} - \text{cS.WIDTH_CENTER_LANE} \]

Transform globalLocale = XYATransform.TiT2(
cS.Locate,
new XYATransform(relativeX,relativeY,Math.PI));
return globalLocale;

Action QuayAreaToQuayCrane

Figure 7.10 depicts the infrastructure used by automatically guided vehicles as they are governed by a QuayAreaToQuayCrane Action. The definition and use
of QuayAreaToQuayCrane is in many ways identical to that of CrossoverAreaToWest, it will not be discussed in detail. The issue of interest is the creation of the contained Move. In the Action-classes discussed above the Move, and its contained FuTrace, is created during construction of the Action. This is not possible for a QuayAreaToQuayCrane: the quay crane moves along the quay and the destination of the FuTrace, which is the locale of the agv-turntable, is only known at the time the Action activated. This means the Move must be created after the Action is activated. How this is achieved is shown in the definition of QuayAreaToQuayCrane, as follows.

```java
class QuayAreaToQuayCrane extends ActionIB {
    PrimeTicket tQC[];
    Move quayParkArea2QuayCrane
    public QuayAreaToQuayCrane(CrossoverScene cS) { ... }
    public void script(int quayParkArea) { ... }
    public void beyond() { ... }
}
```

tQC[]: an array of PrimeTickets created via Semaphores sQC[0-5]. These Semaphores guard the agv-turntables of the six quay cranes.

QuayAreaToQuayCrane(CrossoverScene crossoverScene): constructor, creates the PrimeTickets providing access to the quay cranes.

script: instructs the MFDriver to travel from the quay park area, to the agv-turntable of a quay crane, as follows. In line 02 the CrossoverAgv is requested the integer-id of the destination quay crane. After permission to travel to the agv-turntable is acquired in line 03, the CrossoverAgv is requested where the agv-turntable is in line 04. It is guaranteed the quay crane will not move until the automatically guided vehicle indicates it has arrived upon the agv-turntable in line 08. In line 05 quayArea2QuayCrane is created, the current locale of the quay crane is passed as a parameter, and the contained FuTrace is created accordingly. In line 07 the script is blocked until the finished event of
quayArea2QuayCrane has occurred, this event occurs when the automatically guided vehicle has stopped upon the agv-turntable.

```java
01 crossoverAgv = (CrossoverAgv) actor.getService();
02 int quayCraneID = crossoverAgv.getQuayCraneID();
03 tQC[quayCraneID].insist();
04 XYASpace quayCraneLocale =
05 crossoverAgv.getQuayCraneLocale();
06 quayArea2QuayCrane =
07 new QuayParkArea2QuayCrane(transform, quayCraneLocale);
08 mfd.begin(quayArea2QuayCrane);
09 quayArea2QuayCrane.finished.watch();
10 crossoverAgv.arrivedAtQuayCrane();
```

events tQC[tQC[quayCraneID]].free() upon occurrence of the Move event quayParkArea2QuayCrane.beyond, this is when the tail of the automatically guided vehicle has passed beyond the agv-turntable of the quay crane.

### 7.4 Validation in Mini-World

The crossover transport system is considered a successful application of the AgileFrames software development method: *Moves* prescribe complex automatically guided vehicle movements; *Flags*, *Signs*, and *Signals*, are used to define complex interaction between an agv-MFDriver and its *Actor*; collisions are avoided without introducing needless velocity variations; and advanced traffic behavior is modelled within *Action*. However, not all software components described by the workbench are utilized. The complex traffic rules introduced in section 5.1 are not used, the reason for this is that the crossover layout imposes a one-way traffic flow. The *OrderQueue* introduced in section 3.5 is not used; exclusive access to resources results in *Ticket* method invocation interlacements, which implies an *Actor* can not request a resource before its predecessor. The *Letter* and the *Method-Letter* are not used to implement communication between *Agents*; the use of remote method invocation, as provided by *java.rmi*, is preferred.

In section 2.4 the AgileFrames approach to simulation is discussed. Using a traditional approach, it could occur that the behavior of a ‘simulated’ CrossoverScene may be different from the behavior of a ‘real’ CrossoverScene. This should be avoided, because the behavior of CrossoverScene greatly effects
the performance of a crossover transport system. For this reason, integration of simulation with engineering is important for the development of the CrossoverScene. How this is achieved is discussed below.

validating the layout

In order to validate the shape of virtual lanes of CrossoverScene, they are visualized within Virtuality, which is a 3d-visualization environment provided by the FoRcEs workbench. A SceneIB, the super-class of CrossoverScene provided by the TrAcEs workbench, will visualize its layout in Virtuality, if its getFuTraces method is implemented correctly. CrossoverScene implements getFuTraces by returning all FuTraces defined within the Moves its Actions will generate; the code is shown in appendix A. Using this visualization, the required FuTraces were defined, using a trial-and-error.

After defining the shape of virtual lanes, it must be validated automatically guided vehicles will follow them as intended. Regardless of traffic control by CrossoverScene, collisions can only be avoid if the virtual lanes are defined ‘far enough apart’. This can be verified by defining a simulated automatically guided vehicle that can follow virtual lanes, and visualize its activities within Virtuality. Definition of such an simulator-Agent is easy because the FoRcEs workbench provides the GenericMFDriver, introduced in section 4.3, which can also follow the virtual lanes defined in XYASpace. The ContainerCarryAgvGMFD extends GenericMFDriver; it only adds the behaviour to visualize itself in Virtuality. After creation, a ContainerCarryAgvGMFD will follow the virtual lanes of CrossoverScene, if instructed to do so via Moves; how this is achieved is discussed below. It was verified the virtual lanes are far enough apart, by observing the visualization of ContainerCarryAgvGMFD’s as they follow virtual lanes. This approach is not very reliable, it would be preferred to request Virtuality to compute collisions between the automatically guided vehicles. This advanced approach requires knowledge how to program java-3d [43], and has not been used to date.

In order to instruct a ContainerCarryAgvGMFD to follow the virtual lanes, it is created together with a TrAcEs Actor that must activate a Circulate acquired from the local CrossoverScene. A LoGoS CrossoverAgv must assign the agv-Actor a job to do so. To do this, CrossoverAgvSimulator was developed, which is created with a reference to the agv-Actor, and the ServiceID of the local CrossoverScene. After requesting the agv-Actor the perform the job it becomes passive and simply responds to requests generated by Circulate as follows.

getStackAreaID(int side): returns the int-identifier of the destination stack
area. The side-parameter indicates if the automatically guided vehicle is on the western or the eastern side of the crossover transport system.

getQuayAreaID: returns the int-identifier of the destination quay area. The invocation will block until the automatically guided vehicle may depart, in this way a CrossoverAgy can time the arrival of containers at quay cranes.

getQuayCraneLocale: returns the locale of the quay crane at this time, the local will not change until arrivedAtQuayCrane is invoked.

arrivedAtQuayCrane: informs the quay crane that the automatically guided vehicle is on the agv-turntable of the quay crane of which the locale was previously returned.

toCirculate: returns true; the automatically guided vehicle will circulate forever.

getStackAreaID(int side): returns 0-5, if the automatically guided vehicle is as the western side, and 6-11 of the automatically guided vehicle is at the eastern side.

getQuayAreaID: returns 0-10

getQuayCraneID(int quayParkAreaID): returns the integer-identifier of the quay crane closest to the quay park. The quay crane positions are fixed.

getQuayCraneLocale: returns the fixed locale of the quay crane of which the integer-identifier was previously returned.

arrivedAtQuayCrane: ignored.

validation the scene

The Semaphores of CrossoverScene must guard the shared infrastructure resources, and it must be validated this is indeed the case. To do this a custom 2d-visualization tool was developed. Every CrossoverScene-BasicSemaphore was associated with a rectangle within the 2d-visualization; the size and position of the rectangle corresponds with the resource the semaphore guards. The color of the square would be red if the single unit of Semaphore capacity was depleted, and green otherwise. Using this custom 2d-visualization tool, it was possible 'see' the status of the CrossoverScene at runtime. If only a single ContainerCarryAgyGMFD were active, then the 2d-visualization tool could
be used to see it the activated **Circulate** would insist and free the correct **Tickets**, and at the right time.

**results**

The **ContainerCarryAgvCMFD** has demonstrated that there are some errors in the definition of virtual lanes within the crossover transport system: virtual lanes do not always connect. This is easily observed within **Virtuality**, because the **ContainerCarryAgvCMFD** will follow a virtual lane exactly as it is defined, and will 'jump' a few meters from the end of one virtual lane to the beginning of the next. It is clear these errors must be corrected, but it is also clear that the **ContainerCarryAgvCMFD** does not display true automatically guided vehicle behavior, which will not 'jump'. Furthermore, it is not equipped with trace intelligence as described in section 4.2, and it does not take the driving characteristics of a real automatically guided vehicle into account. For this reason, it may be true virtual lanes should be defined further apart.

The custom 2d-visualization tool for the **CrossoverScene** has demonstrated that an activated **Circulate** does insist and free the correct **Tickets**, but that the timing of free is sometimes too early. This is usually caused by a beyond-event that is defined a half agv-length beyond the end of a **FuTrace**, instead of a full agv-length. This mistake is easily made because a **FuTrace** is defined with respect to the center of an automatically guided vehicle, as is demonstrated in the following example. Suppose an automatically guided vehicle must travel to a parking area that is approximately the size of an automatically guided vehicle. If so, the **FuTrace** leading to the destination will end in the center of the parking area. The distance from the center of the parking area to its exit is half an agv-length, but a parking area is free after an automatically guided vehicle has travelled its full length out of the parking area.

**further work**

The **CrossoverScene** of the crossover transport system is intended to be deployed as a real traffic control system, and must therefore be able to deal with real problems, such as mechanical and electrical failures. For this reason, the **CrossoverScene** should be used to control traffic generated by mini-agvs. But, mini-agvs were developed after the **CrossoverScene**, and this test has not been conducted to date. However, the mini-agvs were themselves tested using **DemoScene**, which is a relatively simple traffic control system. **DemoScene**
was confronted with real world problems, such as total failure of the mini-
agv, and with failure of the computer network. These failures are discussed
below.
Chapter 8

Implementation Issues

During the development of AgileFrames, and during the implementation of mini-world, decisions regarding the implementation of the AgileFrames workbench had to be made. The decision to use object-oriented and multi-threaded software technology is considered obvious. Component-oriented modelling, as prescribed for Java beans [45], is also considered a primary design approach, in particular the event source-listener pattern. Regarding distributed programming, sockets are considered a low-level communication mechanism that is transparent to a programmer using Java™ remote method invocation. Furthermore, Java™ provides remote polymorphism, allowing a new degree of freedom when developing distributed control systems. In section 8.1 it is indicated how AgileSpace is implemented, and which technologies it relies upon. In section 8.2, it is motivated why TrAcEs Agents are distributed over the network as they are. In section 8.4, it is shown how transactions are used to implement the CollectTicket in order to reserve a set of PrimeTickets in an atomic manner. In section 8.5, it is shown how nondeterministic selection of PrimeTickets by the SelectTicket is implemented using callback.

8.1 Implementation of AgileSpace

AgileSpace is intended as simplified communication infrastructure for distributed Agents. Users are presumed to be traffic control engineers, without a background in software engineering, for whom current distributed programming techniques may be too complex. In general, AgileSpace only provides a simplified form of java-rmi [42], java-jini [37], and java-spaces [41]. However, the author advises traffic control engineers to use the advanced technologies
directly, because the simplified forms do not allow compile-time checking, which is a serious threat to reliable software development. Furthermore, if traffic control engineers wish to engage in automation, it is best they learn to use the required tools.

**Letters**

Exchange of Letters is implemented as exchange of Entries via a JavaSpace.

**Methods**

Methods invoke a method upon a server-Agent that has registered itself with AgileSpace. Such a registration is stored as an entry in the java-space. Whenever a Method is sent, it is first tested to see if the server-Agent has been registered. If the server-Agent has been registered, then a reference to the server-Agent is acquired via the registration entry in the java-space, and the method is invoked upon the server-Agent, if it is implemented. To do this, reflection technology is used, which is provided by the java language itself in the java.lang.reflect package. If the method is not implemented by the server-Agent, then the Method will throw an Exception indicating this is the case. If the server-Agent has not been registered, then the Method will throw an Exception indicating this is the case.

**Signs**

Signs are only implemented for use in a non-distributed context, and are only used for interaction between a TrAcEs Actor and a FoRcEs FDriver. A Sign’s watch method corresponds with an Object’s wait method, and a Sign’s send method corresponds with an Object’s notifyAll method. A Sign does extend the Object with new behavior: after send, watch is no longer blocking.

**Signals**

Signals are only implemented for use in a non-distributed context, and are only used for interaction between a TrAcEs Actor and a FoRcEs FDriver. Upon invocation of send, a Signal invokes the specified method upon the Object passed during creation. To do this, reflection technology is used.
Services

Advertisement and lookup of Services is implemented as registration and lookup of service-proxies with the jini lookup service [46].

8.2 Distribution of Traffic Control Agents

The AgileFrames design method introduces four types of Agents: the Actor, the Scene, the MFDriver, and the Service-Agent. As long as Agents only interact by passing each other simple data, such as numbers and records, it is not an issue where an Agent is hosted, and decisions regarding Agent distribution can be determined according to operational constraints imposed by the configuration of the computer network. The Java programming language allows Agents to interact by not only passing simple data, but also by passing each other behavior: class definitions may be downloaded by a client-Agent from a server-Agent. So called remote polymorphism truly revolutionizes distributed programming. However, it does not change the need for resolving implementation decisions. Regarding the Actor, the Scene, and the MFDriver, the following three issues are clear:

- the MFDriver resides upon a mobile computer platform embedded within the handling machine it must control; this is simply required to implement real-time control. Within the context of real-time control, it also seems best that the trace intelligence contained within Moves is also interpreted within the name-space of the MFDriver.

- a Scene resides upon a fixed computer platform embedded within the environment and will be remote with respect to other Scenes and MFDivers.

- the interaction between an Actor and its MFDriver is intense and long-term, and it is best both are hosted together upon the mobile computer platform embedded within the handling machine.

Regarding activation of Actions, which are maintained within the context of a Scene, the two following scenarios are considered.

activation within the Scene: upon invocation of createAction, only a stub will be downloaded to the Actor; execute will be invoked remotely upon the Action still residing within the context of the Scene. Upon invocation of mfdriver.begin(move) within the context of an Action, the Move
will be downloaded to the MFDriver; the interactions between Actions and Moves will be remote. Interaction between Tickets and Semaphores will be local, but the interaction between super-Actions and sub-Actions will be remote. Because Tickets may be passed between Actions this implies the interactions between Tickets and Actions must also be considered remote.

**activation within the Actor:** upon invocation of createAction, the Action and its contained Moves, will be downloaded to the Actor; execute will be invoked upon the Action as it resides within the context of the Actor. There is no need to download Moves to the MFDriver upon invocation of mfDriver.begin(move), because both already reside within the same context. Interaction between Tickets and Semaphores will be remote, but the interaction between super-Actions and sub-Actions will be local because all are downloaded into the same context. The interactions between Tickets and Actions will also be local.

It is best to minimize remote interaction. For any Action, the number of interactions between PrimeTickets and Semaphores is approximately the same as the number of interactions between Actions and Moves, implying no preference for either approach. CollectTickets and SelectTickets do introduce extra, and complicated, interactions and it would be best if these are not complicated further by the possibility of remote failures. For this reason, it seems best to activate the Actions within the context of the Actor. Also, downloading of Actions should be preferred within the context of Scene failure. If a Scene fails then the downloaded Action will contain the Exception-handling code required to deal with the situation, possibly an emergency Actor-to-Actor traffic control procedure to navigate out of the area guarded by the failed Scene.

### 8.3 Distribution of Semaphores

Many different implementations for semaphores exist within many different systems at this time. For example, the Sun's Solaris operation system provides System V IPC-semaphores, thread semaphores, and posix semaphores. TrAcEs is concerned with synchronizing distributed Actors with respect to each other via Semaphores and therefore remote access to Semaphores is required. The issue of remote semaphores is widely addressed within the context of distributed shared memory systems. For example, DIPC [47] is an experimental distributed version of IPC providing remote access to IPC semaphores, and an efficient implementation of remote access to semaphores.
is proposed by Ramachandran and Singhal [48, 49]. TrAcEs is similar to these systems in the sense that it provides remote access to Semaphores, and in this section it is discussed how this is achieved. TrAcEs is different to these systems in the sense that it introduces *atomic reserve* with respect to a set of Semaphores that may also be distributed with respect to each other, this is the subject of section 8.4.

The BasicSemaphore provided by TrAcEs is defined using synchronized java-methods. Java guarantees such methods are only executed in an atomic manner. All methods defined upon a BasicSemaphore are synchronized, in this way it is trivial to implement the original semaphore methods *p* and *v* introduced by Dijkstra. However, due to the decision to define the PrimeTicket and the Semaphore as remote the insist method defined upon the PrimeTicket is not forwarded directly to a P method upon the BasicSemaphore, a callback approach is used instead. In this way the Actor-Thread may experience a RemoteException, is minimized. The BasicSemaphore is implemented as follows.

```java
class BasicSemaphore implements Semaphore { ...
    private int currentCapacity;
    synchronized public boolean attempt(PrimeTicket pT) { ... }
    synchronized public boolean reserve(PrimeTicket pT) { ... }
    synchronized public void free (PrimeTicket pT) { ... }
    synchronized private void select () { ... }
}
```

*boolean attempt(PrimeTicket pT)*: assigns capacity to the PrimeTicket immediately, or not at all. If capacity is assigned, then `currentCapacity` is decremented with the requested amount and the result is *true*. Otherwise, the value of `currentCapacity` remains unchanged and the result is *false*.

*boolean reserve(PrimeTicket pT)*: reserves capacity for the PrimeTicket. If the requested capacity can be assigned immediately, `currentCapacity` is decremented with the requested amount and the result is *true*. Otherwise, the value of `currentCapacity` remains unchanged, the PrimeTicket is pushed into a reservingQueue, and the result is *false*.

*void free(PrimeTicket pT)*: reclaims capacity granted to a PrimeTicket. The value of `currentCapacity` is incremented with the claimed capacity, and the Semaphore-thread waiting in method `select` is notified.
void select(): pulls PrimeTickets from the front of reservingQueue as long as
the value of currentCapacity exceeds the claim of the PrimeTicket. If so,
the the BasicSemaphore decrements currentCapacity with the requested
amount, and invokes the remote method assigned upon the PrimeTicket
in order to inform its claim has been honored. If not the Semaphore-
thread waits until notified via free.

The PrimeTicket presents two methods which are equivalent to Dykstra's P
and V methods, insist and free. Invocation of the free method is simply
forwarded to the BasicSemaphore. Invocation of the insist method results
in an invocation of reserve upon the BasicSemaphore, if the result is false
the Actor-Thread is forced to wait until the Semaphore-Thread calls back via
invocation of assigned, otherwise the Actor-Thread does not block and the
invocation of insist terminates.

The BasicSemaphore is equipped with an extra Thread which loops within the
select method and is responsible for invoking the remote method assigned
upon PrimeTickets. This approach is preferred over an alternative where the
Thread invoking free to also used to invoke assigned upon PrimeTickets. This
alternative is inferior because the Thread invoking free, which originated from
a particular Actor, may become involved in problems of some other Actor. The
implementation presented above ensures the Thread invoking free will return
to its Actor as soon as possible with the least chance of failure.

It is important that an automatically guided vehicle can not continue on its
way without permission. For this reason the PrimeTicket is implemented as
fail-safe, it will not allow an invocation of insist to de-block before callback
by the Semaphore. In this way, failure of a Semaphore will not result in collisions.

transmission latencies

As stated above, the interaction between a BasicSemaphore and its PrimeTicket
is remote and implemented utilizing java remote method invocation. The
BasicSemaphore does not take any transmission latency issues into account.
Suppose a BasicSemaphore bS has created PrimeTickets tA and tB and that
tA.reserve is invoked before tB.reserve. It may occur that bS.reserve(tB) is
invoked before bS.reserve(tA), due to an extreme transmission latency.

Potential deadlocks due to transmission latencies must be dealt with by the
traffic control engineer. To this aim AgileFrames introduces the OrderQueue,
which is not implemented at this time. Note that the latency issues do not
imply ticket method invocations are not atomic. Transmission latencies im-
ply that the order in which a Semaphore processes atomic method invocations is non-deterministic.

8.4 Atomic Reserve

The CollectTicket can reserve a set of PrimeTickets in an atomic manner. In this section it is shown how this is achieved using two phase transactions [50]. First, two alternative approaches are discussed and rejected.

A possible approach is to deny concurrency between Actors. If so, an Actor is only de-activated upon execution of a synchronized statement, and only re-activated upon de-activation of another Actor. This means an Actor reserving a sequence of PrimeTickets is the only Actor doing so. Such an approach is only acceptable if all Actors reside upon the same computer. In fact, within such a context, possibly within an event-scheduled simulation, this approach would be desirable. Unfortunately, it is not desirable to reside all Actors within the same computer with respect to scalability of a traffic control system. Furthermore, such an approach has a detrimental effect upon the response time of Actors with respect to their MFDrviers because they may be de-activated at the time a particular event occurs, and it may take a long time before an Actor is re-activated.

Another approach is to determine which Actions modelling traffic intelligence are susceptible to deadlocks. One can identify Semaphore-sets which must be guarded by a BasicSemaphores with a single unit of capacity. An Action must lock the Semaphore-set by acquiring the single unit of capacity, before it may reserve capacity from the Semaphores in the set. Determining Semaphore-Sets that are potentially involved in a deadlock is difficult. In [51] it is shown how concurrent java programs may be verified for deadlock, and this means such sets may be discovered at compile-time. However, such verifications may take a long time to compute. Furthermore, this approach would require the traffic control engineers to agree upon the use of particular BasicSemaphores in order to avoid deadlock. This approach is rejected because two phase transactions allow traffic control engineers to dynamically lock Semaphore-sets without the need to statically define a BasicSemaphore upon which everyone must agree.

two phase transactions

A reserve-set is a set containing PrimeTickets which must be reserved in an atomic manner. Atomic reserve is achieved if all the PrimeTickets within a
reserve-set are pushed into the `reservingQueue` of their respective Semaphores during a particular time-window within which the Semaphores involved do not push any other PrimeTickets into their `reservingQueue`. To do this, Semaphores are defined as *transaction participants* that can reserve PrimeTickets under a *transaction*. After a Semaphore is requested to reserve a PrimeTicket under a transaction, it will deny reserving any other PrimeTicket under a different transaction, until it is requested to commit or abort the reservation. A group of Semaphores reserving PrimeTickets under the same transaction, will all be requested to commit the reservation, or all will the requested abort the reservation. The transaction is coordinated by the transaction-Object itself.

Consider the code presented below. In lines 03 to 05, three Semaphores are requested to reserve, under a transaction Object called `txn`, PrimeTickets they created a earlier. During the request, the Semaphores register with the `txn` as transaction participants, which means they implement the methods `prepare`, `commit`, and `abort`. In line 06, the `txn` is requested to drive the transaction to success or to failure. First, the `txn` requests all three the Semaphores, which registered as transaction participants before, to `prepare`. During `prepare`, a Semaphores checks if it is possible to push the PrimeTickets that are reserving under the transaction into its `reservingQueue`. If so, then the Semaphore will return `COMMIT`, and `ABORT` otherwise. If all Semaphores return `COMMIT`, then the `txn` invokes `commit` on all participating Semaphores, after which line 06 terminates normally. If any Semaphore returns `ABORT`, then the `txn` invokes `abort` on all participating Semaphores, after which line 06 terminates abnormally with an Exception.

```java
primeTicket[1] = semaphore[1].createPrimeTicket(1);
primeTicket[2] = semaphore[2].createPrimeTicket(1);
primeTicket[3] = semaphore[3].createPrimeTicket(1);

// 01 Transaction txn = AgileSpace.createTransaction();
02 try {
03    semaphore[1].reserve(primeTicket[1], txn);
04    semaphore[2].reserve(primeTicket[2], txn);
05    semaphore[3].reserve(primeTicket[3], txn);
06    txn.commit();
07    // atomic reserve succeeded
08 }
09 catch(Exception e) {
10     // atomic reserve failed, try again later.
11 }
```

In general, the Exception will indicate that none of the statements in lines
03 to 05 have been executed and the the atomic reserve may be tried again later. In extreme situations it may occur that the Exception indicates that the status of the failed two-phase transaction is not known. If so, an external monitor must be requested to correct the problem.

The CollectTicket is a transaction client which requests Semaphores to reserve PrimeTickets under a two-phase transaction; its reserve method is implemented with code similar to that presented above.

deadlock detection

As stated above, determining Semaphore-sets that are potentially involved in the deadlock of activated Actions is a non-trivial task inherent to distributed Actors. There exists a Communicating Sequential Processes [52] interpretation of java-threading [51], and research is being conducted for CSP verification of java for livelock and deadlock detection. This verification mechanism might be useful in detection of potential deadlocks between Actions.

arbitrary nesting

The traffic control engineer may wish to arbitrarily nest PrimeTickets, SelectTickets, and CollectTickets, within a CollectTicket. At this time a CollectTicket may only nest PrimeTickets, but to achieve arbitrary nesting the following approach may be used. The top CollectTicket will acquire a transaction via AgileSpace and request all its sub-Tickets to reserve under the transaction. If the sub-Ticket is a PrimeTicket, it will reserve with the Semaphore under the transaction. If the sub-Ticket is a SelectTicket or a CollectTicket, they will request their sub-Tickets to reserve under the transaction passed by the super-Ticket. In this way all the PrimeTickets referred to by the top-CollectTicket, either directly or indirectly via the sub-Tickets, will reserve under the same transaction. The top-CollectTicket will then request the transaction manager to drive the transaction to success or failure.

8.5 Nondeterministic Selection

A SelectTicket can insist upon the first PrimeTickets from a set which will acquire its claim first. The SelectTicket can not know which PrimeTicket will acquire its claim first, and in this section it is demonstrated how nondeterministic selection of PrimeTickets is implemented. The basic idea is simple,
the first sub-Ticket to call back to the SelectTicket after its claim has been assigned is the selected sub-Ticket, after which the SelectTicket invokes free upon all the other non-selected sub-Tickets. However, this simple idea introduces complex problems. First is the problem caused in combination with arbitrary nesting, where it may occur that the selected PrimeTicket is freed because it occurs twice as a sub-Ticket. The second problem concerns a potential low-level multi-threading deadlock during the call-back procedure. In order to avoid these problems, arbitrary nesting is not implemented at this time.

arbitrary nesting

Suppose there exists an extra-wide automatically guided vehicle that requires two parking areas to wait for a container exchange in an stack area with three parking areas. If so, then the automatically guided vehicle must park in the first and second parking areas, or in the second and third parking area. This demonstrates that it may be desired to select between two distinct CollectTickets referring to a common sub-PrimeTicket. In order to achieve arbitrary nesting, the following approach may be used. The SelectTicket monitors sub-Tickets utilizing a callback mechanism, upon assignment of its claim a sub-Ticket will invoke assign(int i) upon the SelectTicket. CollectTickets may also be nested within a SelectTicket, this implies that the CollectTicket must also monitor its sub-Tickets and use a callback mechanism after all its sub-Tickets are assigned. The first sub-Ticket to callback upon a SelectTicket is the selected sub-Ticket, the SelectTicket utilizes this invocation to invoke abort upon all other non-selected sub-Tickets, in order to inform them their reservation is no longer required and that they may free themselves. A nested SelectTicket and a nested CollectTicket will forward the abort request to their sub-Tickets. Whenever a SelectTicket nests CollectTickets or SelectTickets that refer to a common sub-Ticket, then the common sub-Ticket will receive multiple abort requests. However, such common sub-Ticket will also have received multiple requests to callback upon assignment. A Ticket must therefore only free itself after it has received as many requests to abort as it has received requests to callback.

multi-threaded deadlock

The Semaphore-Thread that makes a call-back upon a PrimeTicket is utilized to effectuate the callback by the PrimeTicket to its super-Ticket. Furthermore, the Semaphore-Thread that invokes assign(i) upon the SelectTicket is
also used to invoke abort upon all the other sub-Tickets. It may occur that two Semaphores honor the claim of two PrimeTickets involved with the same SelectTicket at approximately the same time. Two Semaphore-Threads call-back up the Ticket-tree, but only one can invoke assign(i) upon the SelectTicket first, it is called the winner-Thread. After selection, the winner-Thread moves down the Ticket-tree from the SelectTicket to the PrimeTickets via invocations of abort. The loser-Thread is still moving up the Ticket-tree. Threads moving in opposite directions through a datastructure in which they lock objects, as is the case described above, may deadlock upon each other. Thread deadlocks due to mutual object locking are very tricky to understand an possibly even more difficult to resolve.
Chapter 9

Conclusion

In the introduction of this thesis, several questions were asked regarding the development of a safe, flexible, agile, and scalable, automated transport system. Have these questions been addressed and answered?

Safety

- The use of sensors in order to maintain distance between automatically guided vehicles is not a primary issue in this thesis. In section 2.1 references to research in this field are provided.

- Synchronization of concurrent traffic agents in order to avoid collisions is a major issue in this thesis. Section 5.1 introduces the Semaphore as a traffic control software-component that may be used to guard shared infrastructure components. A Semaphore can synchronize concurrent and distributed Actors via PrimeTickets, a traffic control software-component that can be downloaded from a Semaphore to a remote Actor.

Avoiding needless decelerations is an important issue in this thesis, because this would impair high-performance. Section 4.2 introduces the Flag as a traffic control software-component that can be used to indicate when an automatically guided vehicle must start decelerating in order to avoid a possible collision on a shared infrastructure, such as an intersection. If permission to use the shared infrastructure resource is granted immediately by the Semaphore that guards it, then deceleration is not required.

Traffic control engineers remain responsible for programming traffic control logic that is safe. Traffic control engineers are also responsible
for developing traffic control logic that is efficient, but combining safety and avoiding needless velocity decelerations is a confusing issue, from a sequential programming point of view. In section 6.2 it is shown why this is so, and how concurrent programming techniques can be used to develop safe and efficient logic for Actions in an intuitive manner.

- The use of sensors to avoid injuring personnel is not addressed in this thesis. However, the use of sensors in order to maintain distance between automatically guided vehicles is a related field. Furthermore, material handling systems are required to address safety issues, and industrial automatically guided vehicles should be equipped with some kind of safety mechanism and be able to perform an emergency-stop.

- Use of an automatically guided vehicle must be fail-safe. Section 4.2 introduces the Precaution as a traffic control software-component that can be used to indicate how a MFDriver must decelerate in order to avoid a possible collision on a shared infrastructure, such as an intersection. As long as the Actor does not remove the Precaution the MFDriver will not use the shared infrastructure. If network communications between an Actor and a Scene fails, then the Semaphore can not inform an Actor’s PrimeTicket that its request to use a resource is granted. An Action, programmed by a traffic control engineer, and downloaded by an Actor from a Scene, should not remove a Precaution before a Semaphore has informed the PrimeTicket that its request has been granted.

If the Scene’s computer fails then communication between an Actor and a Scene fails as described above. If the computer hosting the Actor and the MFDriver fails, the machine itself must be programmed to stop in a fail-safe manner. This issue is beyond the scope of this thesis, but is relevant with respect to the mini-agv. In section 6.4 it is discussed how the mini-agv’s mechatronics bring the mini-agv to a full-stop when the mini-agv’s embedded computer fails.

Traffic control engineers are only human, and may introduce programming errors that compromise safety. In section 9.2 below, an envisioned traffic control software tool is described that can generate software components, called basic-Actions, that use a shared infrastructure in a safe and efficient manner.
Flexibility

- Selection of alternative routes in order to avoid traffic congestions is an important issue because it enhances high-performance. Section 6.1 is concerned with this issue only, and in particular: anticipation, resource (route) selection, collecting (infrastructure) resources, unstructured interaction with other agents, passing resources between actions, preparation, and concurrent activation.

- Avoiding deadlocks is a field of research in its own right, and methods for discovering deadlocks is beyond the scope of this thesis. However, in this thesis two tools for avoiding deadlocks after they are discovered, possibly during simulations, are presented. In section 3.5 the OrderQueue is presented, and in section 6.2 it is shown how the OrderQueue may be used to avoid deadlocks caused by transmission latencies. In section 6.1 the CollectTicket is presented and it is shown how to avoid deadlocks caused when Actors try to reserve (collect) the same (infrastructure) resources.

- Dealing with exceptions is essential. In section 4.2 the Flag is presented, it may be used to define events a MFDriver experiences. In general, Flags define events that are required to monitor regular progress of the MFDriver, but Flags may also be used to define exceptional events. In section 6.3 it is shown how Flag-events, regular or exceptional, may be propagated up the Action-tree. It may be so that failure of the MFDriver does not create an exceptional condition, but that it only stops progressing. If so, then the MFDriver will stop generating any Flag-events. Flags are monitored as Signs, and in section 3.3 it is shown how Signs are monitored under a time-out. It may also be true that the MFDriver itself experiences an Exception, possibly due to a programming error. If so, then the MFDriver may abort Flags. In section 3.3 it is shown how Signs are monitored under an Exception.

The mini-agv is a real automatically guided vehicle, and it is subject to electrical and mechanical failures. In section 6.4 it is shown how such exceptions are dealt with in mini-world. It is also shown how the mini-agv, its FoRcEs-MFDriver, and its TrAcEs-Actor, may be removed from a transport system, with minimal impact upon ongoing transport activities.

- Scenes are implemented in a distributed manner. If Scenes are hosted on different computers, then failure of one computer does not prevent
Scenes hosted on other computers from controlling traffic, and transport will continue as long as it is feasible and safe.

Agility

- Traffic-layout design is grafted upon control of automatically guided vehicles by defining virtual lines using FuTraces in XYASpace, which are also functional specifications of what an automatically guided vehicle must do. A traffic control engineer remains responsible for understanding how a MFDriver will interpret a FuTrace.

- At this time, it is not possible to automatically generate Actions that contain the traffic logic for simply following a virtual lane. A traffic control engineer is required to develop such an Action, which it tedious an repetitious work. In section 9.2 below, it is discussed how this task may be automated.

- A Scene's definition is a software-class, and may be used to create any number of Scene-Agents that all govern an identical infrastructure, but at different locations.

- Scene-Agents provide a traffic control Service to the transport system. In section 5.2, it is shown how an agent may advertise its Service. If Actors regularly update their Scene-Service relationships, a Scene may be replaced with minimal impact upon ongoing transport activities.

  If a Scene-Agent is replaced by a new version, an Actor using such a new Scene will automatically download the new types of Actions the new Scene provides using remote polymorphism. In this way Scene-Agents may be replaced with minimal impact upon ongoing transport activities.

- The MoveDriver, which is a sub-Agent within the MFDriver, is a generic component that can compute the best straight line and acceleration in order for the MFDriver to progress along a FuTrace. This generic component facilitates rapid development of machine control.

Scalability

The TrAcEs traffic control design method is scalable because the interactions between Actors and Semaphores is localized via a Scene. This issue is discussed in section 5.2.
9.1 Validation of FoRcEs and TrAcEs

In section 1.3 three issues regarding control of machines were discussed.

- The first issue is specifying a continuous sequence of desired states indicating how a machine must progress through the environment. A FuTrace through a FuSpace is such a specification, it also defines a velocity profile specifying how fast the machine must progress through the environment. FuTraces through XYASpace define virtual lanes through a flat surface environment that an automatically guided vehicle may follow.

- The second issue is controlling a machine in order to make it do whatever is specified. In section 4.2 a generic algorithm for computing the best straight line, from a machine's current functional state, to a point on the FuTrace to be followed through any FuSpace. Another generic algorithm computes the best acceleration in order to reach the desired velocity. These generic algorithms are contained within the MoveIB software component.

- The third issue is specifying event-oriented interaction between the machine's controller and other controllers. In section 4.2 the Flag is introduced to specify events that the MFDriver may experience, and it may be used to implement communication from the MFDriver to the Actor. The Flag-Sign allows a Flag to be monitored by 'watching' for its occurrence. The Flag-Signal allows a Flag to be monitored via asynchronous event notification, which is an important design strategy in software engineering. In section 4.3 the MFDriver interface is introduced, and in section 4.2 the Precaution is introduced. These software components may be used to implement interaction from the Actor to the MFDriver. The MFDriver interface allows the Actor to pass Moves to the MFDriver, and it allows the Actor to interrupt, resume, and cancel, MFDriver activities. The Precaution allows itself to be 'removed' by the Actor, after which the MFDriver need no longer decelerate in order to stop the automatically guided vehicle in front of a particular (infrastructure) resource.

FuTraces, Flags, and Precautions, are all packaged together in a Move, which functions as a traffic control object for a machine. In order to validate the MFDriver-architecture and the assumption that a Move possesses enough expressive power to control a machine, a keyboard-sized automatically guided
vehicle has been developed within the context of mini-world. This validation is discussed in section 4.3.

The interaction between the MFDdrivers and the Actors of the crossover transport system, depicted in figure 2.3, is defined utilizing Moves. How this is achieved is the subject of chapter 7, the result demonstrates that the Move does indeed possess enough expressive power to model intricate interactions between a traffic control system and automatically guided vehicles.

In [28, 30, 31] it is shown that TrAcEs facilitates development of flexible and safe automated traffic control systems. In this thesis, the case study of the crossover terminal demonstrates this once again, but then with added detail regarding the interaction between MFDdrivers and Actors. In this thesis, the ABCDEScene and its sub-Scenes demonstrate that TrAcEs also facilitates agile and scalable traffic control systems.

Further Validation of FoRcEs

There already exist control systems for automatically guided vehicles. Frog Navigation Systems (FNS) provided the agv-control system for the automatically guided vehicles transporting containers on the ECT-DSL container sea-terminal, and for the experimental automatically guided vehicles of the OLS test-site. There are significant differences between the FNS agv-control system and the FoRcEs approach, in particular the definition of a FoTrace as a continuous function. It would be interesting to port the FoRcEs-based mini-agv control system to an experimental automatically guided vehicles for the OLS test-site in order to compare both approaches.

An Improved Development Cycle

The development of the CrossoverTerminal demonstrates how simulation may be integrated in the development cycle, whilst still implementing code that is suitable for deployment is the real traffic control system.

9.2 Rapid Development of Traffic Control

The AgileFrames workbench should provide tools that assist the logistic engineer with the complex task of developing an automated logistic system. The following tools could be provided for FoRcEs.
9.2. RAPID DEVELOPMENT OF TRAFFIC CONTROL

A Layout Tool: could assist in the creation of a layout by visual programming, similar to powerpoint or autocad. This tool produces a layout class which, upon instantiation, creates a set of FuTraces defining the layout at a particular position and orientation.

An Infrastructure Tool: could assist in the definition of shared resources, according to the layout-class and the physical dimensions of the machines utilized using the layout. The tool will compute all possible intersections of machine volumes as these follow FuTraces. The computed intersection volumes may be very small, and therefore the traffic control engineer must group intersection volumes into shared resources and associate them with a capacity and traffic rules. The tool produces an infrastructure class which, upon instantiation, creates the Semaphore model guarding the shared resources.

A Move Tool: could generate Moves according to the layout class, the infrastructure class, and the physical dimensions of the machines. For any machine following a FuTrace, it can be computed which shared resources it must utilize. The tool will produce a set of basic-Moves which define how machines utilize individual shared resources. A basic-Move contains a FuTrace which is a segment of the original FuTrace defining the layout. The standard Flag-fields remain null.

The following tools could be provided for TrAcEs.

A Scene Tool: could assist in the definition of Scenes. To do this, the engineer must first indicate how the Semaphores, defined within the infrastructure class, must be organized within a hierarchy. The tool may visualize the Semaphores by displaying the volume of the shared resource it guards, if any. The traffic control engineer may then group neighboring Semaphores together into an infrastructure model which will be maintained by a Scene.

The scene tool can generate a basic Action governing a basic-Move associated with the shared resource guarded by the Semaphore. The basic-Move's standard started-Flag and finished-Flag, will be associated with EvolutionFlagSignals associated with the beginning and the end of the contained basic FuTrace, respectively. The standard basic-Move's ready-Flag will be associated with a TimedFlagSignal associated with the end of the contained basic-FuTrace. The minimal deceleration value is a property of the machine that must perform the basic-Move. The standard basic-Moves beyond-Flag will be associated with an EvolutionFlagSignal.
defined a machine-length beyond the end of the contained basic-FuTrace. The length value is a property of the machine. Upon creation, the basic-Action will acquire a PrimeTicket from the Semaphore, if the Semaphore is not a BasicSemaphore then the engineer must indicate how the claim is formulated. Upon activation, the basic-Action will insist upon the PrimeTicket before instructing the MFDriver to perform the Move. Upon occurrence of the beyond-Flag the PrimeTicket is freed.

Basic Traffic Control Tool: could assist in the generation of scene-Actions for every route within the Scene considered relevant. For any machine following a FuTrace a corresponding sequence of basic-Actions must be activated via execute. The traffic control engineer must indicate which sequences are relevant with respect to the traffic a Scene governs. For every relevant sequence the tool will generate an scene-Action that activates the sequence of basic-Actions. The standard ‘started’ event of the scene-Action is associated with the started event of the first basic-Action of the sequence. The standard ‘ready’, ‘finished’, and ‘beyond’ events of the basic scene-Action are associated with the corresponding events of the last basic-Action of the sequence.

Basic scene-Actions implement so-called basic traffic control, they avoid collisions but do not display any advanced traffic behavior such as anticipation, selection, and collection. The traffic control engineer may extend a scene-Actions and provide a new implementation for its execute method which does implement advanced traffic behavior.

9.3 Societal Implications

Automation of transport systems has largely been restricted to industrial areas, without a noticeable impact upon everyday life. If transport automation does become widespread this will change.

Tendering of Machines for Material Handling Systems

Within a material handling system different types of machines, produced by different manufacturers, may be utilized. It is best if the owner of a material handling system can request many different producers to tender for sale of machines. Such a tender could be specified in terms of a FuSpace, and possibly a set of Moves, prescribed by the owner of the material handling system. Machine manufacturers must develop a machine, together with a
MFDriver, that can follow arbitrary FuTraces within the FuSpace, or at least perform the set of prescribed Moves. If the price is right, their machine will be bought.

Outsourcing Transport Services

To date, an automated transport system is considered the property of the material handling system. However, automated transport is not the core-activity of a material handling business, and it seems reasonable that such a business may wish to outsource its automated transport services. In this way, such companies could lease automatically guided vehicles whenever the need to use them arises. TrAcEs makes outsourcing possible because a TrAcEs traffic control system defines a clear interface itself and the machines it must govern: the Move. The Scenes providing traffic control always remain the property of the material handling system.

The Betuvelijn

The Betuvelijn is an 3 billion Euro project in the Netherlands to construct a new train track from Rotterdam to Germany. This new train track is considered essential for further development of Rotterdam as the largest harbor in Europe, but there are doubts concerning the return on investment. For this reason, it is considered how the Betuvelijn may be used more effectively. A possible solution is to allow other transport modalities to use the new infrastructure, such as automatically guided vehicles.

TrAcEs and FoRcEs could play an important role in opening the Betuvelijn to other transport modalities. First of all, a Betuvelijn exploitation organization could define a FuSpace to represent the Betuvelijn, and Moves for using the Betuvelijn. Any transport vehicle equipped with a MFDriver that can interpret such Moves may use the Betuvelijn. Of course, TrAcEs Actions will model safe, flexible, and agile use of the Betuvelijn. If such multi-modal exploitation is indeed a success, then the traffic control system can be scaled to govern traffic along other infrastructures as well.

Intelligent Infrastructures

At this time, high-end luxury passenger cars, costing approximately 100,000 Euros, may provide driver assistance such as radar-based cruise/distance control, and GPS-based navigation assistance. If an accident occurs, the vehicle
will automatically phone the emergency services. Fully automated, sensor-based vehicle guidance seems feasible, and it seems increased automation of vehicle guidance would improve safety on public roads.

Fully automated, sensor-based vehicle guidance does not need an intelligent infrastructure to guide the vehicle. However, interacting with an intelligent infrastructure could improve safety even further. An intelligent infrastructure could also be equipped with sensors that gather traffic data, and such information may be transmitted to vehicles. For example, an intelligent infrastructure may provide regular updates of a vehicle's position, and the position of other vehicles in the vicinity, or information regarding congestions, and advice how to avoid them. Such information could be processed by a vehicle automatically, or the vehicle could forward the information to the driver via a sound system. Note: it is not required that a vehicle is equipped with equipment to interact with an intelligent infrastructure, nor is it required that a vehicle follows instructions, it is simply better to do so.

An intelligent infrastructure may even facilitate introduction of transport automation because vehicles need not be equipped with expensive sensors and special sensor-data processing computers, it need only be equipped with a mobile communications set, similar to a cell-phone, and a normal computer that is linked to the sound system and acceleration/steering actuators. The infrastructure itself will become more expensive, with cameras and special sensor-data processing computers, and high capacity communications facilities.

An intelligent infrastructure must be worth its money. For roads that are seldom used this may not be the case, at least not in the beginning. For roads with structural congestion problems the extra investment may be worthwhile, especially if a government would reduce taxes for vehicles that can interact with intelligent infrastructures, facilitating their introduction. Taxes may be reduced even further if vehicles behave as instructed by an intelligent infrastructure. If all vehicles, or at least most of them, are capable of interacting with an intelligent infrastructure, then the intelligent infrastructure could also function as an agent that could control traffic in order to avoid congestions, greatly improving road capacity.

The cost of communications equipment and computers, and the issue of mixing automated vehicles with non-automated vehicles, is not the biggest problem regarding the introduction of an intelligent infrastructure. The biggest problem is the format of the intelligence. It is paramount that information can be exchanged between the traffic control agents constituting the intelligent infrastructure and the traffic agents using roads. The current world-
wide-web of the internet demonstrates how difficult this issue is. Different vendors of web-browsers decided they all knew best how to format web-data. As a result, none of the web-browsers can view all pages. It would be unacceptable if the government were to invest in an intelligent infrastructure, only to discover that a vehicle manufacturers decided to market incompatible vehicles. At the same time, it would be unacceptable if vehicle manufacturers were told by the government what is best for automatic vehicle guidance.

Remote polymorphism could be the solution to this problem, because decisions regarding format is reduced to the selection between computing platforms. Well known computing platforms are Microsoft-Windows, Sun-Java, and Linux, but a custom platform may be selected by the automotive industry. Furthermore, vehicle manufactures must agree on a generic interface for controlling vehicles. A possible interface could be the Move defined by FoRcEs. If intelligent infrastructures are ever to become reality, and their services actually provided for real vehicles, then participants in the automotive industry must agree, as soon as possible, on a computing platform and a generic vehicle control interface. This will be to their own advantage.
Appendix A

Code Excerpts

In chapter 7, code snippets, and sometimes pseudo-code, of the CrossoverScene is presented. In this appendix, an overview of the code implementing the CrossoverScene is presented. The CrossoverScene was the first traffic control system implemented using the java-based TrAcEs and FoRcEs workbenches. It was completed in September 2000 [53], by Herman Wierenga, using a prototype version of the workbenches. Since that time, version 1.0 of the workbenches have been developed, and this version is described in this thesis. In order to use the CrossoverScene in this thesis, it has been re-implemented using TrAcEs workbench 1.0. Unfortunately, the GenericMFDriver has not been ported to FoRcEs workbench 1.0 to this date. As a result, it was not feasible to re-implement the the CrossoverScene-MoveS as well. The definition of CrossoverArea2West presented in section 7.3, is how it would have been if the original code had been ported to FoRcEs workbench 1.0. In this appendix, the original code is presented on page 208.
CrossoverScene constants

The crossover layout depends on 21 constants, these are defined as follows.

```java
public interface CrossoverTerminal {
    /** width of all the lanes on the terminal */
    public final float WIDTH_DRIVING_LANE = 5.5f;
    /** width of all the centerlanes on the terminal */
    public final float WIDTH_CENTER_LANE = 6.5f;
    /** length of all the stacklanes on the terminal */
    public final float LENGTH_STACKLANE = 17.5f;
    /** width of all the stacklanes on the terminal */
    public final float WIDTH_STACKLANE = 6.5f;
    /** space on the entrance of all the stacklanes on the terminal */
    public final float FREE_STACK_ENTRANCE = 8.5f;
    /** space on the exit of all the stacklanes on the terminal */
    public final float FREE_STACK_EXIT = 8.5f;
    /** space on both sides of each stack. */
    public final float FREE_SPACE_BETWEEN_STACKS = 6.5f;
    /** space on the entrance of all the parklanes on the terminal */
    public final float FREE_PARK_ENTRANCE = 7.5f;
    /** length of all the parklanes on the terminal */
    public final float LENGTH_PARKLANE = 17.5f;
    /** width of all the parklanes on the terminal */
    public final float WIDTH_PARKLANE = 6.5f;
    /** space on the exit of all the parklanes on the terminal */
    public final float FREE_PARK_EXIT = 10.6f;
    /** space between the southern quaylane and the quay. */
    public final float FREE_QUAY = 22.8f;
    /** space between the western sidelane and the end of the terminal. */
    public final float FREE_TERMINAL_LEFT = 10.5f;
    /** space between the eastern sidelane and the end of the terminal. */
    public final float FREE_TERMINAL_RIGHT = 10.5f;
    /** space between the inner side lanes and the outer stacks. */
    public final float FREE_SPACE_AT_SIDES = 27f;
    /** turn radius of the vehicles in corners of the terminal */
    public final float TURN_RADIUS = 11f;
    /** distance between the cenetr and the turning point of an AGV */
    public final float AGV_TURNING_POINT = 0f;
    /** assumed length of an AGV */
    public final float AGV_LENGTH = 20.0f;
    /** space on the exit of all the turntables on the terminal */
    public final float FREE_TABLE_EXIT = 5.3f;
    /** height of all the turntables on the terminal */
    public final float TABLE_HEIGHT = 14;
    /** width of all the turntables on the terminal */
    public final float TABLE_WIDTH = 28;
}
```
CrossoverScene

The CrossoverScene implements the CrossoverTerminal in order to have access to the constants. During construction, a CrossoverScene not only creates the BasicSemaphores it requires according to the number of stack, but also the Moves its Actions will govern. The reason for this is that CrossoverScene extends SceneIB which requires implementation of getFuTraces in order to upload a generic Scene-proxy that will visualize all FuTraces in Virtuality.

```java
public class CrossoverScene extends SceneIB
    implements CrossoverTerminal {

    public static XYATransform locale;
    public int numberOfStacks;

    public CrossoverScene( 
        String name, 
        XYATransform locale, 
        int numberOfStacks, 
        ServiceID serviceID)
    {
        super(name + "@CrossoverScene");
        this.numberOfStacks = numberOfStacks;
        this.cT = new CrossoverTerminal(numberOfStacks);
        this.serviceID = serviceID;
        this.locale = locale;
        ...
        createSemaphores();
        createMoves();
    }

    public Semaphore[][] sSPA
    public Semaphore[][] sSPAX
    public Semaphore[]  sPCN
    public Semaphore[]  sTCN
    public Semaphore[]  sTCS
    public Semaphore[]  sPCS
    public Semaphore[][] sQPAX
    public Semaphore[][] sQPA
    public Semaphore[]  sQC

    private void createSemaphores() {
        for (int stackAreaID=0; stackAreaID<12; stackAreaID++) {
            for(int parkAreaID=0; parkAreaID<4; parkAreaID++) {
                if (stackNr<12-1) {
                    semPCN[stackNr][parkLane] = new BasicSemaphore(1);
                }
            }
        }
    }
```
public MoveIB[] stackArea2SouthToQuayArea;
public MoveIB[] stackArea2West;
public MoveIB[] crossoverArea2West;
public MoveIB[] crossoverArea2East;
public MoveIB[] crossoverArea2QuayAreaWest;
public MoveIB[] crossoverArea2QuayAreaEast;

private void createMoves() { ... }
    for (int stackAreaID=0; stackAreaID<12; stackAreaID++) {
        crossoverArea2West = new crossoverArea2West(this, stackAreaID);
    }

FuTraces[] getFuTraces() {
    for (int stackAreaID=0; stackAreaID<12; stackAreaID++) {
        fuTraces[j] = crossoverAreaToWest[i].getFuTrace(); j++;
    }
    return fuTraces
}

Note that CrossoverScene does not create a QuayArea2QuayCrane, because these are created during activation of a QuayAreaToQuayCrane.
Action Circulate

public class Circulate extends ActionIB {
    private CrossoverAgv crossoverAgv;
    public StackAreaToWestToQuayArea [] [] stackAreaToWestToQuayArea;
    public StackAreaToSouthToQuayArea[] [] stackAreaToSouthToQuayArea;
    public StackAreaToEastToQuayArea [] [] stackAreaToEastToQuayArea;
    private int stackAreaID;
    private int stackParkAreaID;
    private int quayCraneID;

    public Circulate(CrossoverScene crossoverScene) {
        super(crossoverScene);
        this.cS = crossoverScene;
        for (int stackAreaID=0; stackAreaID<12; stackAreaID++) {
            for(int stackParkAreaID=0; stackParkAreaID<4; stackParkAreaID++) {
                for (int quayAreaID=0; quayAreaID<12-1; quayAreaID++) {
                    if (quayAreaID<=stackAreaID-3) {
                        stackAreaToWestToQuayArea[stackAreaID]
                        [stackParkAreaID]
                        [quayAreaID] =
                        new StackAreaToWestToQuayArea(cS,stackAreaID,stackParkAreaID,quayAreaID);
                    }
                }
            }
        }
    }

    public assimilate(Actor actor, Action super) {
        for (int stackAreaID=0; stackAreaID<12; stackAreaID++) {
            for(int stackParkAreaID=0; stackParkAreaID<4; stackParkAreaID++) {
                for (int quayAreaID=0; quayAreaID<12-1; quayAreaID++) {
                    if (quayAreaID<=stackAreaID-3) {
                        stackAreaToWestToQuayArea[stackAreaID]
                        [stackParkAreaID]
                        [quayAreaID].
                        assimilate(actor,this)
                    }
                }
            }
        }
    }

    protected void script() {
        enterToStackArea();
        while ( toCirculate() ) {
            stackAreaToQuayCrane();
            ...
            quayCraneToStackArea();
        }
        stackAreaToExit();
    }

    public boolean toCirculate() { return crossoverAgv.toCirculate(); }
public void stackAreaToQuayCrane() {
    int quayAreaID = crossoverGv.getQuayAreaID();
    int quayParkAreaID;
    if (quayAreaID > stackAreaID+2) {
        // use StackAreaToEastToQuayArea
    } else {
        if (quayAreaID <= stackAreaID-3) {
            stackAreaToWestToQuayArea[stackAreaID][stackParkAreaID][quayAreaID].
                execute();
            quayParkAreaID =
                stackAreaToWestToQuayArea[stackAreaID][stackParkAreaID][quayAreaID].
                getQuayParkAreaID();
        } else {
            // use StackAreaToSouthToQuayArea
        }
    }
    QuayAreaToQuayCrane quayAreaToQuayCrane =
        new QuayAreaToQuayCrane(cs, quayAreaID, quayParkAreaID);
    quayAreaToQuayCrane.assimilate(actor);
    quayAreaToQuayCrane.execute();
    ...
public class StackAreaToWestToQuayArea extends ActionIB {

    private int stackAreaID, stackParkAreaID, quayAreaID;
    private int quayParkAreaID = -1;
    private StackAreaToWest stackAreaToWest;
    private CrossoverAreaToWest[] crossoverAreaToWest;
    private CrossoverAreaToQuayAreaWest crossoverAreaToQuayAreaWest;
    private CrossoverScene cS;

    public StackAreaToWestToQuayArea(
            CrossoverScene cS,
            int stackAreaID,
            int stackParkAreaID,
            int quayAreaID) {
        super(cS);
        this.cS = cS;
        this.stackAreaID = stackAreaID;
        this.stackParkAreaID = stackParkAreaID;
        this.quayAreaID = quayAreaID;
        stackAreaToWest = new StackAreaToWest(cS, stackAreaID, stackParkAreaID);
        crossoverAreaToWest = new CrossoverAreaToWest[cT.numberOfStacks];
        for (int nr = (stackAreaID - 2); nr >= (quayAreaID + 2); nr--) {
            crossoverAreaToWest[nr] = new CrossoverAreaToWest(cS, nr);
        }
        crossoverAreaToQuayAreaWest =
                new CrossoverAreaToQuayAreaWest(cS, quayAreaID);
    }

    public void assimilate(Actor actor) {
        stackAreaToWest.assimilate(actor);
        for (int nr = (stackAreaID - 2); nr >= (quayAreaID + 2); nr--) {
            crossoverAreaToWest[nr].assimilate(actor);
        }
        crossoverAreaToQuayAreaWest.assimilate(actor);
    }

    protected void script() {
        SelectTicket quayParkAreaST = new SelectTicket(
                (PrimeTicket) crossoverAreaToQuayAreaWest.getAccessTicket(0),
                (PrimeTicket) crossoverAreaToQuayAreaWest.getAccessTicket(1),
                (PrimeTicket) crossoverAreaToQuayAreaWest.getAccessTicket(2),
                (PrimeTicket) crossoverAreaToQuayAreaWest.getAccessTicket(3),
                (PrimeTicket) crossoverAreaToQuayAreaWest.getAccessTicket(4)
            );
        quayParkAreaST.insist();
        quayParkAreaID = quayParkAreaST.getSelectedIndex();
    }
stackAreaToWest.execute();
for (int nr=(stackAreaID-2); nr>(quayAreaID+2); nr--) {
    crossoverAreaToWest[nr].execute();
}
crossoverAreaToQuayAreaWest.execute();

public int getQuayParkAreaID() { return quayParkAreaID; }
}
Action CrossoverAreaToWest

The definition of CrossoverAreaToWest as presented in section 7.3 differs with respect to the actual definition presented below. The difference is a single statement within the constructor, and concerns the creation of CrossoverArea2West. In the actual definition below this statement is as follows.

```java
crossoverArea2West = (CrossoverArea2West)
    cS.crossoverArea2West[stackAreaID].clone(this);
```

The move is not created new, it is created as a clone from an instance of CrossoverArea2West that is maintained by the CrossoverScene. The result of the two approaches is identical.

```java
public class CrossoverAreaToWest extends ActionIB {

    private int stackAreaID;
    private CrossoverArea2West crossoverArea2West;
    private PrimeTicket primTicExtraCenter;
    private PrimeTicket primTicExtraNorth;
    private CrossoverScene cS;

    public CrossoverAreaToWest(CrossoverScene cS, int stackAreaID) {
        super(cS);
        this.cS = cS;
        this.stackAreaID = stackAreaID;
        primTicExtraCenter =
            new PrimeTicket(this,cS.semCenterNorth[stackAreaID] );
        primTicExtraNorth =
            new PrimeTicket(this,cS.semCenterStayNorth[stackAreaID-1]);
        crossoverArea2West=(CrossoverArea2West)
            cS.crossoverArea2West[stackAreaID].clone();
    }

    protected void script() {
        primTicExtraNorth.insist();
        primTicExtraCenter.insist();
        crossoverArea2West.reset();
        actor.getMFDriver().begin(crossoverArea2West);
        crossoverArea2West.finishing.wait(); // receive the Sign.
    }

    public void finished() { primTicExtraCenter.free(); }
    public void beyond() { primTicExtraNorth.free(); }
}
```
Move CrossoverArea2West

The definition of CrossoverArea2West as presented in section 7.3 differs with respect to the actual definition presented below. The difference is how a CrossoverArea2West is created, and how Flags are created. A CrossoverArea2West is created in two steps: first it is created by a CrossoverScene, after which it is cloned for an Action. With respect to the Actor the result is the same, and the issue will not be discussed further. The created Flags are all EvolutionRules, including the finished-event. Upon creation, an EvolutionRule is passed a reference to this Move, and two integers. Upon occurrence of this event the Flag will invoke event upon this Move, and pass the two integers along. Within event, the two integers are used to determine which event actually occurred, after which the appropriate event-handler is invoked upon the governing CrossoverArea2West Action. Regarding the definition of CrossoverAreaToWest, the CrossoverArea2West is must govern behaves as expected. Or rather, from the point of view of the Action, there is no difference.

public class CrossoverArea2West extends MoveIB {

    public CrossoverArea2West(XYATransform cLocale, int stackAreaID) {
        Transform transform = computeGlobalLocale(stackAreaID);
        this.trajectory = createCrossover$West(transform);
    }

    private Transform computeGlobalLocale(
        XYATransform cLocale,
        int stackAreaID) {
        float relativeX = (stackAreaID+1)*cS STACK_WIDTH_INCL +
            cS FREE.Terminal_LEFT +
            cS FREE_SPACE_AT_SIDES +
            cS WIDTH_DRIVING_LANE ;
        float relativeY = cS.getStackLaneY() -
            cS FREE_STACK_EXIT -
            cS WIDTH.CENTER_LANE ;
        Transform globalLocale = XYATransform.T1T2( cS.Locale,
            new XYATransform(relativeX,relativeY,Math.PI));
        return globalLocale;
    }

    private Trajectory createCrossover$West(Transform transform) {
        Trajectory[] composedTrajectory = new Trajectory[1];
        composedTrajectory[0] =
            new GoStraight(CrossoverScene.cT STACK_WIDTH_INCL);
        composedTrajectory[0].setTransform(new POSTransform(0,0,0));
        Trajectory crossover$West = new Trajectory(composedTrajectory);
crossover$West.setTransform(transform);
crossover$West.obstacleAtEnd = false;
return crossover$West;
}

public void createRules(){
    float agvLength = CrossoverScene.cT.AGV_LENGTH;
    float tEnd = trajectory.domain;
    Move.EvolutionRule finishing,finished,beyond;
    finishing = new
    Move.EvolutionRule(this,trajectory,tEnd-agvLength,FINISHING);
    finished = new
    Move.EvolutionRule(this,trajectory,tEnd-agvLength/2,FINISHED);
    beyond = new
    Move.EvolutionRule(this,trajectory,tEnd+agvLength/2,BEYOND);
    this.rules = new Rule[] { finishing,finished,beyond };
}

CrossoverAreaToWest crossoverAreaToWest;

private final int FINISHING = 0;
private final int FINISHED = 1;
private final int BEYOND = 2;

public Object finishing = new Object(); // functions as a Sign

    /**
     * handles events defined by rules,
     * events forwarded to handler defined upon superAction
     */
    public synchronized void event(int eventID,int seqNum) {
        switch (seqNum) {
            case FINISHING:
                this.finishing.notify(); // send the Sign.
                break;
            case FINISHED:
                crossoverAreaToWest.finished();
                break;
            case BEYOND: // stayNorth
                crossoverAreaToWest.beyond();
                break;
        }
    }
Action StackAreaToWest

```java
public class StackAreaToWest extends ActionIB {
    PrimeTicket tCenterStayNorth;
    PrimeTicket[] tStackExit;
    PrimeTicket tCenterNorth;

    public StackAreaToWest(  
        CrossoverScene cS,int stackAreaID,int stackParkAreaID) {
        int index = 0;
        for (int exit=cT.lanesPerStack/2;exit<=cT.lanesPerStack;exit++){
            tStackExit[index] =
                new PrimeTicket(this,cS.semStackExit[stackAreaID-1][exit]);
            index++;
        }
        for (int exit=0;exit<=stackParkAreaID;exit++) {
            tStackExit[index] =
                new PrimeTicket(this,cS.semStackExit[stackAreaID][exit]);
            index++;
        }
        tCenterNorth =
            new PrimeTicket(this,cS.semCenterNorth[stackAreaID-1]);
        tCenterStayNorth =
            new PrimeTicket(this,cS.semCenterStayNorth[stackAreaID-2]);
    }

    protected void script() {
        tCenterStayNorth.insist();
        CollectTicket collectTicket = new CollectTicket(tStackExit);
        collectTicket.insist();
        tCenterNorth.insist();
        leaveStackForCenterWest.reset();
        actor.getMFDriver().begin(leaveStackForCenterWest);
        leaveStackForCenterWest.finishing.wait();
    }

    public void beyondStackArea() {
        for (int i=0;i<tStackExit.length;i++) {
            if (tStackExit[i]!=null) { tStackExit[i].free(); }
        }
    }

    public void finished() { tCenterNorth.free(); }

    public void beyond() { tCenterStayNorth.free(); }
}
```
Action StackAreaToEast

public class StackAreaToEast extends ActionIB {

    protected void script() {
        tCenterStaySouth.insist();
        CollectTicket collectTicket = new CollectTicket(tStackExit);
        collectTicket.insist();
        tCenterNorth.insist();
        tCenterSouth.insist();
        leaveStackForCenterEast.reset();
        actor.getMFDriver().begin(leaveStackForCenterEast);
        leaveStackForCenterEast.finishing.wait();
    }

    public void beyondStackArea() {
        for (int i=0;i<tStackExit.length;i++) {
            if (tStackExit[i]!=null) { tStackExit[i].free(); }
        }
    }

    public void beyondCenterNorth() { tCenterNorth.free(); }

    public void finished() { tCenterSouth.free(); }

    public void beyond() { tCenterStaySouth.free(); }
}
Bibliography


Summary

*Controlling Automated Traffic Agents* concerns a control-software development method for automated control of materials handling systems, in particular for automated transport systems within industrial environments. It is shown how collisions may be avoided without needless velocity variations, and it is shown how to model advanced traffic behavior such as anticipation, route selection, and resource collection. Furthermore, it is shown how an automated transport system using automatically guided vehicles may be scalable and agile.

The availability of computer and communication technology has facilitated the automation of materials handling systems, and introduced the problem of controlling automated traffic. The technology itself also provides a solution: *semaphores*, which are used to avoid ‘collisions’ between concurrent programs. In this thesis it is shown how semaphores may be used to avoid collisions between autonomous traffic agents. The automatically guided vehicle is possibly the most exciting autonomous traffic agent to deal with. However, within the generic context of controlling materials handling systems, vehicle traffic is only intended as a metaphor.

The design of software for controlling an automated transport system is a difficult task; it must be *safe, flexible, agile, and scalable*. Time and costs will be saved by utilizing a control-software development method. The method’s *software architecture* describes the interaction between *software components*. The behavior of software components is implemented in a software language and are the ‘core’ of a method’s *software workbench*.

Research and development by the department of Transportsystems and Logistics at the Technical University in Delft has produced a control-software development method called AgileFrames. The AgileFrames architecture introduces a *network of automated logistic agents* and a software workbench supporting this architecture in the Java programming language. AgileFrames subdivides a control-software system into three complementary sub-systems; a LoGoS sub-system responsible for long-term logistic control, a TrAcEs sub-
system responsible for short-term traffic control, and a FoRcEs sub-system responsible for real-time machine control.

The primary subject of this thesis is the development of distributed traffic control for automated materials handling systems according to the TrAcEs software design method. Controlling machines utilized within an automated material handling system according to the FoRcEs software design method is the secondary subject of this thesis. A short description of FoRcEs and TrAcEs sub-systems is presented below.

**A FoRcEs sub-system:** a sub-network of *machine function driver agents* responsible for performing flow and storage of goods. A machine function driver controls a materials handling machine in real-time, according to a sequence of *moves*.

**A TrAcEs sub-system:** a sub-network of *actor agents* and *scene agents* responsible for controlling flow of traffic within a materials handling system. A hierarchy of scene agents is responsible for controlling traffic within a particular geographical area and maintains a set of *semaphores* which spawn a *virtual infrastructure* and a set of *actions*. An action models the traffic intelligence required to navigate through a virtual infrastructure. An actor agent is responsible for navigating through the virtual infrastructure within the context of a job. To do this it will acquire an action corresponding with the job from the appropriate scene agent. Upon activation by the actor, an action will generate a sequence of *moves* for its machine function driver to perform.

The development of LoGoS logistic control sub-system, is not a central issue within this thesis. It is simply assumed there exist LoGoS agents that generates jobs. In particular, algorithms and data structures required for effective and efficient planning within the context of service provision is beyond the scope of this thesis.

Development of TrAcEs and FoRcEs required a laboratory in which to study transport systems. Consequently *mini-world* was developed, a facility that includes 10 *mini-agvs*, which are keyboard sized automatically guided vehicles. Within this laboratory the TrAcEs and FoRcEs architectures were validated and their respective workbenches implemented. The architectures were validated by implementing several transport systems, most important of which is the *crossover transport system*.

Regarding simulation, AgileFrames only simulates external agents. The mini-agv simulates an external vehicle agent; a FoRcEs or TrAcEs agent interacting with a mini-agv is not aware it is a simulator. The development of
transport systems within the context of mini-world demonstrates how the AgileFrames approach to simulation works in practice.
Samenvatting

Het Besturen van Geautomatiseerd Verkeer

Controlling Automated Traffic Agents betreft en methode voor het ontwikkelen van programma's voor het besturen van systemen die geautomatiseerd goederen verwerken, in het bijzonder geautomatiseerde transportsystemen. Er wordt getoond hoe botsingen voorkomen kunnen worden zonder overbodige variatie van snelheden. Er wordt getoond hoe intelligent gedrag zoals anticipatie, route selectie, en verzamelen, gemodelleerd kan worden. Verder wordt er getoond hoe een geautomatiseerd transport systeem schaalbaar en aanpasbaar kan zijn.

De beschikbaarheid van computers en communicatienetwerken heeft ontwikkeling van geautomatiseerde systemen die goederen verwerken mogelijk gemaakt, en het probleem van geautomatiseerde verkeersgeleiding doen ontstaan. De technologie voorziet ook in een oplossing: semaforen, die gebruikt worden om 'botsingen' tussen parallele programma's te voorkomen. In dit proefschrift wordt getoond hoe semaforen gebruikt kunnen worden om botsingen tussen geautomatiseerde voertuigen te voorkomen. Het geautomatiseerde voertuig is wellicht de meest interessante verkeersparticipant, maar binnen de algemene context van technische automatisering wordt het verkeer van voertuigen alleen gebruikt als metafoor.

Het ontwerpen van besturingsprogramma's voor geautomatiseerde transportsystemen is moeilijk: deze systemen moeten veilig, flexibel, aanpasbaar, en schaalbaar, zijn. Er zal tijd en geld bespaard worden als er een ontwikkelingsmethode gebruikt wordt. De architectuur van zo'n methode omschrijft de interactie tussen besturingscomponenten. Het gedrag van besturingscomponenten wordt vastgelegd in programma-objecten die de 'kern' van een programma-werkbank voor de methode vormen.

Onderzoek bij de vakgroep Transport Techniek en Logistieke Techniek bij de Technische Universiteit in Delft heeft een ontwikkelingsmethode opgeleverd
voor bestuuringsprogramma's die AgileFrames heet. De architectuur van AgileFrames introduceert een netwerk van logistieke participanten en een programma-werkbank die deze architectuur ondersteunt in de programma taal Java. AgileFrames verdeelt een geautomatiseerd besturingsysteem onder in drie complementaire sub-systemen: een LoGoS sub-systeem verantwoordelijk voor lange-termijn logistieke besturing, een TrAcEs sub-systeem verantwoordelijk voor korte-termijn verkeersgeleiding, en een FoRcEs sub-systeem verantwoordelijk voor continue besturing van machines.

Het belangrijkste onderwerp van dit proefschrift is het ontwikkelen van gedistribueerde verkeersgeleiding voor systemen die geautomatiseerd goederen verwerken volgens de TrAcEs methode. Het secundaire onderwerp is het ontwikkelen van besturingen voor machines volgens de FoRcEs methode. Een korte omschrijving van deze TrAcEs en FoRcEs sub-systemen volgt hieronder.

**Een FoRcEs sub-systeem:** een sub-netwerk van machine-functie bestuurders die verantwoordelijk zijn voor transport en opslag van goederen. Een machine-functie bestuurder bestuurt een machine continu, volgens een serie van ‘bewegingen’.

**Een TrAcEs sub-systeem:** een sub-netwerk van begeleiders en verkeersagenten die verantwoordelijk zijn voor het besturen van verkeer binnen het systeem dat goederen verwerkt. Een hiërarchie van verkeersagenten is verantwoordelijk voor het besturen van verkeer binnen een bepaald gebied. Hiertoe beheren zij een verzameling semaforen die een virtuele infrastructuur vormen, en een verzameling acties. Een actie bevat de intelligentie die een begeleider nodig heeft om een voertuig door de infrastructuur te loodsen. Zodra een begeleider een transportopdracht krijgt, moet deze de ‘actie’ behorende bij de opdracht bemachtigen van een verkeersagent. Zodra de begeleider de ‘actie’ activeert zal deze een serie ‘bewegingen’ genereren voor de machine-functie bestuurder.

Ontwikkeling van een LoGoS sub-systeem voor logistieke besturing is geen onderwerp van dit proefschrift. Het wordt verondersteld dat zo'n sub-systeem bestaat en dat het zinnige opdrachten voor begeleiders genereert. Procedures en data-structuren ten behoeve van planning worden niet in dit proefschrift behandeld.

De ontwikkeling van TrAcEs en FoRcEs benodigde een laboratorium om voortuigssystemen te bestuderen. Hiertoe werd mini-wereld ontwikkeld, een faciliteit met 10 mini-agv’s, hetgeen automatisch bestuurde voertuigen zijn
ter grootte van een toetsenbord. Binnen dit laboratorium werd de TrAcEs en FoRcEs besturingsmethoden gevalideerd en de programma-werkbanken geïmplementeerd. Ter validate werden verschillende geautomatiseerde transportsystemen ontwikkeld, waaronder het oversteek transport systeem.

Betreffende simulatie, AgileFrames simuleert alleen externe entiteiten. De mini-agv simuleert een normaal geautomatiseerd voertuig, maar de TrAcEs begeleiders, the TrAcEs verkeersagenten, en de FoRcEs machine-functie bestuurders zijn hiervan niet op de hoogte. De ontwikkeling van transportsystemen in mini-wereld demonstreert hoe deze aanpak ten opzichte van simulatie werkt in de praktijk.
About The Author

The author of this thesis was born on June 12, 1968, in Delft, The Netherlands. In 1986, after receiving his high-school diploma from the "Stanislas College" in Delft, he started a study in Computer Science at the "Delft University of Technology". In 1994 he received his M.Sc degree, after which he drafted for one year of "civil service", and then worked free-lance for one year providing internet services. In 1996, he started as a Ph.D. student at the "Delft University of Technology", and this thesis is a result of his research.
Glossary

**Agile:** the ability to adapt to changing customer requirement in an expeditious manner with low cost.

**Agent:** an entity with a particular responsibility. Within a virtual world an automated control system, such an entity is an **Object**, possibly equipped with a **Thread**.

**AGV:** automatically guided vehicle.

**AGV-Trace:** trace within **XYASpace** defining a virtual lane automatically guided vehicles may follow.

**Asynchronous Event Notification:** notification of an event by asynchronous invocation of a method, by the source-**Agent**, upon the listener-**Agent**.

**Automated:** control is carried out by, or largely supported by, computer facilities.

**Business Logic:** logic responsible for decisions within a logistic control system. In general, the software code found in the extension of predefined software components that provide it with the behavior required within a business.

**Class:** a Java class defines fields, which are variables, and provides implementations for methods.

**Concurrent:** at the same time, in parallel.

**Context:** random access memory within a computer within which an object or **Agent** resides. In computer science, it may also be called a **namespace**.

**Distributed Processes:** processes that are physically separate. In general, distributed programs are executed on separate computers.
**Flexible:** the ability to adapt to changing operational circumstances immediately.

**Infrastructure:** sub-division of the environment into infrastructure components such as roads, intersections, and parking lots.

**Virtual Infrastructure:** an infrastructure equipped with information technology for the purpose of assisting traffic agents.

**Interface:** a group of Java *method declarations*. A Java class the implements an interface must provide an implementation of all methods declared in the interface. An interface is also a type. A Java class may implement multiple interfaces.

**Layout** the structure of a material handling system. In this thesis, layout usually refers the the traffic layout, which is a set of virtual lanes automatically guided vehicles may follow.

**Logic:** program code.

**Materials Handling System:** a network of agents for performing the flow and storage of goods.

**Mechatronics:** the electrical interface to a machine, used to set actuators, and to read sensors.

**Network Communication Middleware:** software used to implement communication between distributed programs. Well-known communication middleware is java-rmi, omg-CORBA, and microsoft-DCOM.

**Object:** if typeset as *object*, it is an Java information object. An object's type is determined by its class and the interfaces that the class implements.

**Traffic Control Engineer:** a person responsible for developing automated traffic control. It is assumed such a person uses the FoRcEs and TrAcEs software development methods.

**Real-Time Control:** immediate control.

**Remote Polymorphism:** object-oriented polymorphism within a distributed context; requires dynamic loading of classes, via the computer network, at run-time.
Software Architecture: a design approach for developing software. Object-oriented software architectures specify how software objects must behave. Component-oriented software architectures specify the behavior of objects according to properties and events.

Software Development Method: software architecture, supported by an object library.

Synchronized: timed with respect to each other. In normal speech, synchronization usually refers to doing something at the same time. In computer science, synchronization refers to not doing something at the same time.

Thread: a light-weight program within a larger program. Concurrent Threads share memory and may access common variables within the larger program.

Virtual World: the information space spawned by a network of computers. Within such a space information objects reside and programs are executed.

Virtual Infrastructure: an Semaphore model of shared resources, it resides within the virtual world.

Virtuality: 3d-visualization environment provided by the FoRcEs workbench.
The Netherlands TRAIL Research School for Transport, Infrastructure and Logistics

A series of theses:

A Knowledge-based Concept Exploration Model for Submarine Design
C.G.J.M. van der Nat, March 1999, T99/1
TRAIL Thesis Series, Delft University Press, The Netherlands

The Maritime Pilot at Work: Evaluation and Use of a Time-to-boundary Model of Mental Workload in Human-machine Systems
F.C. van Westrenen, May 1999, T99/2
TRAIL Thesis Series, Eburon, The Netherlands

Quantitative Analysis of Shipping Markets
A.W. Veenstra, April 1999, T99/3
TRAIL Thesis Series, Delft University Press, The Netherlands

Supported Driving: Impacts on Motorway Traffic Flow
M.M. Minderhoud, July 1999, T99/4
TRAIL Thesis Series, Delft University Press, The Netherlands

Multiclass Continuum Modelling of Multilane Traffic Flow
S.P. Hoogendoorn, September 1999, T99/5
TRAIL Thesis Series, Delft University Press, The Netherlands

Driving with Intelligent Vehicles: Driving Behavior with Adaptive Cruise Control and the Acceptance by Individual Drivers
M. Hoedemaeker, November 1999, T99/6
TRAIL Thesis Series, Delft University Press, The Netherlands

Technology Assessment of Automated Vehicle Guidance - Prospects for Automated Driving Implementation
V.A.W.J. Marchau, January 2000, T2000/1
TRAIL Thesis Series, Delft University Press, The Netherlands
On Classes of Min-max-plus Systems and their Applications
Subiono, June 2000, T2000/2
TRAIL Thesis Series, Delft University Press, The Netherlands

Operational Control of Internal Transport
J.R. van der Meer, September 2000, T2000/5
TRAIL Thesis Series, Delft University Press, The Netherlands

Analytical Dynamic Traffic Assignment with Interacting User-Classes: Theoretical Advances and Applications using a Variational Inequality Approach
M.C.J. Bliemer, January 2001, T2001/1
TRAIL Thesis Series, Delft University Press, The Netherlands

Time-based logistics: An analysis of the relevance, causes and impacts
G.J. Muilerman, April 2001, T2001/2
TRAIL Thesis Series, Delft University Press, The Netherlands

Layout and Routing Methods for Warehouses
K.J. Roodbergen, May 2001, T2001/3
TRAIL Thesis Series, The Netherlands

Bundeling van infrastructuur, theoretische en praktische waarde van een ruimtelijk inrichtingsconcept
TRAIL Thesis Series, Delft University Press, The Netherlands

Innovation Steps towards Efficient Goods Distribution Systems for Urban Areas
TRAIL Thesis Series, Delft University Press, The Netherlands

Safety analysis of Transport Corridors
N. Rosmuller, June 2001, T2001/6
TRAIL Thesis Series, Delft University Press, The Netherlands
Dynamisch Railverkeersmanagement, besturingsconcept voor railverkeer op basis van het Lagenmodel Verkeer en Vervoer
A. Schaaftsma, October 2001, T2001/7
TRAIL Thesis Series, Delft University Press, The Netherlands

Chains and Networks in Multimodal Passenger Transport. Exploring a design approach
W. Bockstael-Blok, December 2001, T2001/8
TRAIL Thesis Series, Delft University Press, The Netherlands

The Business of Modularity and the Modularity of Business
M.J.J. Wolters, February 2002, T2002/1
TRAIL Thesis Series, The Netherlands

Planning and Control Concepts for Material Handling Systems
F.A. Vis, May 2002, T2002/2
TRAIL Thesis Series, The Netherlands

Information Architecture and Electronic Market Performance
TRAIL Thesis Series, The Netherlands

Mind the Gap; Bridging Theories and Practice for the Organization of Metropolitan Public Transport
W.W. Veeneman, June 2002, T2002/4
TRAIL Thesis Series, Delft University Press, The Netherlands

Design of multimodal transport networks, a hierarchical approach
TRAIL Thesis Series, Delft University Press, The Netherlands

A Renaissance of Stations, Railways and Cities, Economic Effects, Development Strategies and Organizational Issues of European High-Speed-Train Stations
P.M.J. Pol, October 2002, T2002/6
TRAIL Thesis Series, Delft University Press, The Netherlands
Freight transport: at any price? Effects of transport costs on book and newspaper supply chains in the Netherlands
H. Runhaar, December 2002, T2002/7
TRAIL Thesis Series, Delft University Press, The Netherlands

Connectors. The Way beyond Transferring
S.C. van der Spek, February 2003, T2003/1
TRAIL Thesis Series, Delft University Press, The Netherlands

Controlling Automated Traffic Agents
D.G. Lindeijer, February 2003, T2003/2
TRAIL Thesis Series, Eburon, The Netherlands