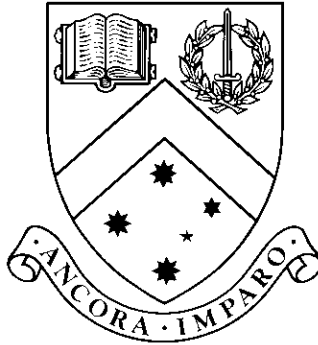


Worst-case time predictions for soft real-time robotic systems

by

Johannes Stickel,



Thesis

Submitted by Johannes Stickel

in partial fulfillment of the Requirements for the Degree of
Bachelor of Computer Science with Honours (1608)

Supervisor: Prof. Heinz Schmidt

Associate Supervisor: Dr. Ian Peake

**School of Computer Science and Software
Engineering
Monash University**

November, 2004

© Copyright

by

Johannes Stickel

2004

Contents

List of Tables	vi
List of Figures	vii
Abstract	viii
1 Introduction	1
1.1 Background	1
1.2 Project Overview	2
1.3 Thesis Structure	3
2 Real-time systems	4
2.1 Introduction	4
2.2 Scheduling theory	5
2.3 Worst-case execution time analysis	6
3 Formal methods	8
3.1 Introduction	8
3.2 Formal models	9
3.3 Finite State Machines	10
3.4 FSA and concurrent systems	11
4 Architecture Description Languages	13
4.1 Introduction	13
4.2 Rich Architecture Description Language	15
4.3 RADL & Finite State Automata	16
5 The system model	18
5.1 Robyn - the ER1 robot	18
5.2 A sample scenario	20
5.3 The model	21
6 Monitoring	23
6.1 Monitoring techniques	23
6.2 Possible points of measurements	25
6.3 Actual Implementation	27

7	Measurements	29
7.1	Overview	29
7.2	Exploratory Data Analysis	30
7.3	Atomic actions	30
7.3.1	Move forward	31
7.3.2	Turns	33
7.3.3	IR sensors	35
7.4	Cell Sequences	37
7.5	The Maze	38
7.6	Monitoring influence on system performance	38
8	Worst-case Execution Time Predictions	40
8.1	Algorithm	40
8.2	Prediction preconditions	41
8.3	Soft worst-case execution times	43
8.4	Predictions	45
8.4.1	Parameters	45
8.4.2	Sequences	46
8.4.3	Maze	47
9	Results & Conclusions	48
9.1	Results	48
9.2	Limitations	49
9.3	Conclusions & Future work	50
	Appendix A The FSA Model	52
A.1	Application	52
A.2	Robot	53
A.3	Orientation	54
A.4	Map	56
	Appendix B Additional Diagrams	58
B.1	Forward movement	58
B.2	Turns	59
B.3	IR sensors	60
B.4	Monitoring influence	62
	Appendix C Deliverables	63
	Appendix D Clarification of Original Contribution	64
	References	65

List of Tables

7.1	Summary of the forward movement execution time measurements . . .	31
7.2	Summary of the turn execution time measurements	33
7.3	Summary of the IR sensor reading time measurements	36
7.4	Summary of the sequence execution time measurements	37
7.5	The maze solving time measurements	38
7.6	Comparison of IR sensor reading times with and without monitoring .	38
8.1	Sample means of action execution times depending on context	42
8.2	Prediction parameters without monitoring	45
8.3	Prediction parameters with monitoring	45
8.4	Monitored sequence predictions	46
8.5	Maze worst-case execution time predictions in seconds	47

List of Figures

5.1	The ER1 robot system overview	18
5.2	The maze used in the experiment	20
5.3	A high-level RADL model of the sample system	21
6.1	Illustration of monitoring possibilities	25
6.2	Illustration of the system monitoring approach	28
7.1	4-plot of the forward movement execution time without monitoring	32
7.2	4-plot of the turn execution time without monitoring	34
7.3	Comparison of IR sensor reading times (depending on break)	35
8.1	Context comparison of forward movements and turn execution times	41
8.2	Context comparison of IR sensor reading times	42
A.1	Application FSA	52
A.2	Motor FSA: A part of the robot component	53
A.3	Sensors FSA: A part of the robot component	53
A.4	Movement part of the orientation component FSA	54
A.5	Forward IR sensor part of the orientation component FSA	54
A.6	Right IR sensor parts of the orientation component FSA	55
A.7	Simplified FSA of the map component	56
A.8	Single map position in detail	57
B.1	Autocorrelation plots of the forward execution times	58
B.2	Comparison of the left and right turn histograms	59
B.3	A autocorrelation plots of the turn execution times	59
B.4	4-plot of the IR sensor reading times (1 s break)	60
B.5	A autocorrelation plot of the IR sensor reading times	61
B.6	Sensor reading packet round trip times in the wireless network	61
B.7	Comparison of monitored and unmonitored atomic actions	62

Worst-case time predictions for soft real-time robotic systems

Johannes Stickel,
johannes@csse.monash.edu.au
Monash University, 2004

Supervisor: Prof. Heinz Schmidt
Heinz.Schmidt@csse.monash.edu.au
Associate Supervisor: Dr. Ian Peake
Ian.Peake@csse.monash.edu.au

Abstract

The number of distributed real-time systems in human life is steadily increasing. Since many of them perform safety-critical tasks, reliability and predictability of these systems is crucial. It is widely accepted that the application of formal methods can help to verify the correctness of such systems. RADL is an architecture description language which can be used to formally describe the architecture of a distributed real-time systems. It aims at support for architectural reasoning about reliability and worst-case execution times.

RADL uses finite state machine models to describe the semantics of the system components and communication protocols. This thesis studies a technique to predict execution times based on finite state machine models. The correctness of the prediction method is examined and a minimal monitoring instrumentation is presented which is used to observe a real system during its execution. This work is conducted in the context of a simple robotics application. The worst-case execution time prediction method was successfully applied to predict the time a mobile robot needs to solve a maze. However, the monitoring instrumentation was shown to be more intrusive than expected.

Worst-case time predictions for soft real-time robotic systems

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Johannes Stickel
November 2, 2004

Chapter 1

Introduction

1.1 Background

Several billion computer processors are manufactured every year. Only a small percentage is used in desktop computers. The vast majority is hidden in modern products from DVD players to vehicles and power plants. These systems have to be highly reliable and often have to guarantee certain response times. If their requirements specify time constraints, they are called real-time systems. How important the reliability of such systems is becomes apparent when the consequences of a failing drive-by-wire system in a car, or a security related controller in a power plant is considered.

Although real-time systems have a long history, they are still difficult and expensive to build. Despite all efforts to make these systems reliable, they are often not as reliable, as we expect them to be. Every few years there is an accident, which is spectacular enough to catch public awareness, for example the Therac-25 accidents[1] in the late 80s, the crash of the European Ariane 5 launcher on its maiden flight in 1996[2] and the loss of NASA's Mars Climate Orbiter only 3 years later. The failure in the last case was caused by the use of different measuring units in the several engineering groups involved in the project.

In many cases, the causes seem to be simple and avoidable. Their roots often lie in wrong assumptions, unexpected conditions and ambiguous specifications. The proponents of formal methods in software engineering point out[3], that many errors could be avoided by the use of formal specifications. Formal specifications are grounded in mathematics and provide unambiguousness. Thus, misunderstandings can be prevented and the specifications are suitable for formal analysis of system properties like liveness and safety. With additional information, formal specification can allow the estimation of non-functional system properties like reliability and worst-case execution time.

Formal methods were perceived to be immature and not scalable enough by practitioners[4]. For a long time, their use in industry was rare. However, the situation changed and formal methods have gained attention in the industry during the last decade. Efforts to bridge the gap between science and industry have been made by both sides. Many companies experiment with formal methods or even use them regularly today. But an integration of formal methods in common development processes is still missing.

Another extensively discussed approach to make software development more efficient and reliable is the building of software by composition of components. Components are tested thoroughly and filed into a library. Software is then developed by choosing appropriate components from a library and configuring them correctly. This is similar to approaches in other areas of engineering. Component-based software development could help to create a real software engineering discipline[5] with all properties associated with the term engineering in general.

The need to describe and configure component-based systems led to the creation of architecture description languages. Most of these languages have a formal semantic foundation and can be considered to be formal methods. *Rich Architecture Description Language* (RADL) is such a language which is also suitable to describe real-time systems. It allows to describe a system as a collection of components and communication channels between the components. The component semantics and the communication protocols are defined by *finite state automata* (FSA). Among other things, an architecture description can be used to analyse non-functional system properties like reliability[6] and worst-case execution time[7].

1.2 Project Overview

This project studies the prediction of worst-case execution times of distributed real-time systems using FSAs. It focuses on robotic systems and uses a mobile robot solving mazes as sample system.

Extending the work by Schmidt et al.[7], this project experiments with the application of a RADL model to predict worst-case execution times of a real system. The main research questions are:

1. Can a RADL model be used to predict the worst-case execution time of a real system?
2. Do measurements of the real system confirm the predicted worst-case execution times?
3. How tight are the predictions? Are the predictions too pessimistic to be useful?

The real system used in this project is the mobile robot Robyn (see section 5.1). Robyn was programmed to solve mazes with a simple and efficient algorithm. The

focus of this project is the study of a method to predict worst-case execution times. Therefore, the robot application is rather basic. A FSA model of Robyn and its environment is created and used to make predictions, how long the robot needs to solve the maze. The measured execution times and predictions are compared and the results are discussed.

1.3 Thesis Structure

The first chapters of the thesis delve into the concepts already mentioned in the introduction:

- Chapter 2 imparts basic knowledge about *real-time systems* and techniques which are related to the analysis of these systems. In particular, schedulability theory and worst-case execution time analysis are treated.
- Chapter 3 provides an review of important *formal methods*. Finite state machines are discussed more closely as they are the foundation of the worst-case execution time prediction method, which is examined in this thesis.
- Chapter 4 surveys several *architecture description languages*. Then RADL, a specific architecture description language using finite state machines as semantic model, is discussed in detail.

The following chapters picture the project on which the thesis is based:

- Chapter 5 introduces a formal *system model* of the sample system used in the project: a mobile robot solving mazes. The model is based on RADL.
- Chapter 6 discusses different *monitoring* techniques which can be used to observe the behaviour of the sample system. Furthermore, the technique used in this project to make execution time measurements is detailed.
- Chapter 7 elucidates the *measurements* conducted in the course of the project. A primary analysis of the data is also provided. This analysis checks some fundamental assumptions about the system.
- Chapter 8 presents *worst-case execution time predictions* based on the measurements described in the last chapter. A comparison with measured execution times and a discussion of the results follows.
- Chapter 9 draws *conclusions* from the previous chapters and summarises the main results of the project. The chapter also attempts to put the project in a broader context.

Chapter 2

Real-time systems

2.1 Introduction

A real-time system is any computer system, in which correctness does not only depend on functionally correct computations, but results also have to be available within certain time constraints. A differentiation is often made between *hard* real-time systems and *soft* real-time systems.

Today, real-time systems can be found everywhere. Common examples of hard real-time systems are areas like avionics, automotive engineering, power plants, medical devices and so on. In each of these cases, a failure of a system to meet its deadlines can cause loss of life or at least an unacceptable economic loss. Therefore many hard real-time systems are often also named safety-critical systems.

Soft real-time systems are even more common and can be found in each modern home. This category includes DVD players and Voice-over-IP applications. A missed deadline will perhaps reduce the quality, but has no serious consequences. If a DVD player occasionally skips a frame, it may even remain unnoticed.

There are several concepts which are closely related with real-time systems. Embedded systems are computer systems specialised for a specific task. They are often expected to work without human interaction and are hidden in a larger device. Another category is reactive systems. These are systems spending their time waiting for external stimuli and reacting on them.

2.2 Scheduling theory

The most important property of a real-time system is its ability to meet its deadlines. The verification of temporal correctness is usually done by a schedulability analysis which requires a preceding worst-case execution time analysis.

Given a set of tasks (including start times, execution times and deadlines) schedulability analysis examines, whether it is possible to schedule these tasks such that all tasks are finished in time. Different scheduling algorithms were developed to determine a correct execution order.

Traditional scheduling algorithms often assume that tasks have to be executed periodically. One of the most discussed and applied algorithms is Rate Monotonic Scheduling[8]. This algorithm assigns the priority of a task according to its request frequency. A task with a higher frequency gets a higher priority. The scheduler always runs the task, which has the highest priority of all ready tasks. Since the priority of a task only depends on its request frequency and the request frequency is fixed, the priority of the task is also fixed. Rate Monotonic Scheduling is optimal in the sense, that if a set of tasks can be scheduled by a fixed priority scheduling algorithm, then it can also be scheduled by the Rate Monotonic algorithm.

However, the algorithm has some problems. The original algorithm works only in an idealised situation. It assumes among other things that tasks do not synchronise. The theory was generalised and many limitations of the original algorithm were overcome. A detailed overview can be found in [9]. But a remaining drawback is a schedulable bound which is significantly less than 100%. The schedulable bound is the maximum CPU utilisation for which schedulability of the tasks is guaranteed. Depending on the number of tasks, the schedulable bound decreases to slightly less than 70%. Another problem is the handling of aperiodic tasks. They can be served in periodic time slots, but this reduces the systems to polling. The system designer has to decide how often the system checks, whether the task should be executed.

Another popular scheduling algorithm is Earliest Deadline First (EDF)[8]. It is a dynamic priority scheduling algorithm. The task with the earliest deadline always gets the highest priority and is therefore executed first. EDF is optimal in the sense that if a set of tasks can be scheduled by any algorithm such that all tasks are finished before their deadline, then EDF can also schedule these tasks correctly. This result assumes, that preemption is allowed. The use of dynamic priority increases the scheduling overhead, but it also admits a 100% CPU utilisation.

EDF works fine, if all tasks are schedulable. But it assumes that all tasks are equally critical. If a vast number of events arrive within a short time, then EDF executes as many random event handlers as possible. Often it is useful to take the importance of the event into account and to process only the most critical events. Other scheduling algorithms which consider the criticality of the tasks in overload situations are also available[10].

2.3 Worst-case execution time analysis

Schedulability analysis requires that the execution times of the tasks are known. Unfortunately, the execution time of a task depends on its parameters and may vary widely. If we consider a trivial implementation of the function $f(n) = n!$, then the execution time of the function depends linearly on n . The worst-case execution time (WCET) is the maximum time a task, function or simply a piece of code may need. In order to ensure a correct schedule, the computation of the worst-case execution times is necessary. This is often a difficult task and in general undecidable due to its equivalence to the halting problem[11]. In the example above, a WCET can be determined by assuming an upper bound on n . With an upper bound, the maximum number of multiplications, comparisons, conditional jumps and decrements is known. If it is also known, how long each of these instructions take on the given hardware architecture, the worst-case execution time of $f(n)$ can be computed. Evidently, the WCET does not only depend on the implementation of the function and the application (in this case the upper bound of n), but also on the target hardware platform.

WCET analysis is the activity of determining WCET estimates. These estimates should be safe and tight. A WCET estimate is safe if the actual worst-case execution time is shorter than the estimate. This property ensures the temporal correctness of the real-time system. But to be useful, the WCET estimate also has to be tight, that is the overestimation should be as low as possible. This is needed since the whole estimated WCET has to be reserved for the task, even if it actually consumes much less time.

Most approaches follow the steps which are sketched above. The first step is an analysis of the program flow. The result of this step is a set of possible execution paths. Many methods demand the programmer to provide appropriate information to make this task easier[12]. If this analysis is done on source code level, the flow information has to be translated to machine language. Compilers which optimise the code can change the program flow, and are therefore difficult to handle. Now the execution time of the paths can be calculated and the longest path can be determined. The calculation of the execution times is complicated by the effects of caches, branch prediction, pipelines and other features of modern processors. While it is possible to ignore the effects of these features, this results in a unacceptable overestimation of the execution times. Therefore it is necessary to model the host processor system.

Engblom[13] identifies three categories of methods to calculate the worst-case execution time given the program flow information:

1. The path-based methods work with explicitly represented paths. The execution times of the different paths are calculated and the longest execution time is determined by comparison of the path execution times. The number of possible execution paths in even very simple programs is often huge. This limits the applicability of these methods.

2. The tree-based methods use a bottom-up approach. They first calculate the execution times of smallest program parts, and reason about the estimates of the bigger parts compositionally. These methods are much faster than path-based methods and confine the execution path explosion problem partially. But it is in general not possible to make tight WCET estimations of the smaller program parts without information of the invocation context. Therefore predictions of WCET by tree-based methods tend to be less tight.
3. Implicit Path Enumeration Techniques (IPET) represent the program flow as a set of algebraic or logical constraints, for example as integer programming (IP) problem. The WCET estimate is calculated by solving the IP problem. IPET solve most problems of the methods above. Although solving IP problems is exponential in general, this happens rarely in practice[14].

WCET analysis of pieces of code is a well-established research area. The focus of this project is slightly different. We take the WCET estimations of the single functions for granted. Instead, the worst-case behaviour of a component-based systems is examined. Although the method looks similar, it addresses another abstraction level. The smallest execution unit is not a machine instruction, but a service request. A path represents a sequence requests between the components of the system. The used method is basically path-based, but tree-based elements could be integrated to increase performance and reduce the number of paths.

Chapter 3

Formal methods

3.1 Introduction

A formal method can generally be defined as a “development method based on some formalism” [3]. A formalism is a formal system that can be used for modelling purposes. Finite state automata, Petri nets and logics are examples of formalisms. They all have in common that they are based on precise, mathematical definitions. As a result, the interpretation of formal models is unambiguous. Beside preventing misunderstandings, formal models are also computer processable. Formal methods use formalisms in order to improve the development process.

There are many “myths” [4] about formal methods. One of them is that they are only used to prove the correctness of a system. Verification is certainly an important application of formal methods. But complete correctness proofs are seldom and costly. Often only small critical parts of a system are proven. Other uses of formal models are more common. An advantage of a formal specifications is the exposure of inconsistencies. Alone the writing process of a formal specification may improve the understanding of the system, since a precise formulation is enforced. Many formal methods also support the simulation of the specified system. Simulations allow experimenting with models of the system and can provide valuable insights in their behaviour. Other uses of formal specifications include the generation of use cases and even the generation of an executable system. Formalisms described in this chapter also provide the foundation for the architecture description language introduced in the next chapter.

This chapter proceeds as follows. First several formalisms are shortly described and the relations between them are discussed. Special attention is paid to state machines, since the worst-case execution time predictions presented later are made using a state machines. Additionally, models specified in other formalisms like temporal logics or process algebras are often analysed with automata theory. This makes automata theory particularly important.

Many formalisms abstract from time which is unfavourable for the analysis of real-time systems. Therefore some extensions are presented that aim at reasoning about temporal correctness. Some analysis techniques are casually introduced and questions about decidability and complexity are raised.

3.2 Formal models

The use of mathematics and symbolic logic to specify and reason about programs has a long history in computer science. Hoare logic[15] was proposed in 1969. It uses Hoare triples (a set of preconditions, a program code fragment and a set of postconditions) to examine the correctness of the code fragment. The code fragment has to guarantee that the postconditions hold after its execution, if the preconditions hold before its execution. In this case, the code fragment is called partially correct. If it also guarantees its termination, it is totally correct. However, Hoare logic is not suited for large program specifications. There are specification languages available, which support the development of complex programs. These include the Z notation[16], which is based on set theory and first-order predicate logic, and its relative B. The latter even assists in developing C code from the specification.

First-order logic can be used to formalise system properties as axioms of a formal theory. Other properties can then be derived as theorems from the axioms. Unfortunately, first-order predicate logic is not convenient to describe real-time systems since it does not directly support the notion of time. Furthermore, first-order theories in general are not decidable. This means that there is no algorithm which can determine, whether a given property can be derived from the system axioms. Temporal logics, like LTL and CTL, have been developed to represent time adequately. They provide temporal quantifiers like “always” and “eventually”. However, these logics can only describe the temporal order of events[17]. Special temporal logics exist to reason about real-time systems. Logics of this kind can also specify the temporal distance between the events. One example of this category is Real-Time Temporal Logic.

Another possibility to specify a system is the use of a process algebra. Process algebras were developed to study parallel or distributed systems by algebraic means[18]. Well-known representatives are CSP, CCS and ACP. To specify a system, a set of processes and actions between them are defined. Processes are defined by a composition of simpler processes, while the simplest process does not have any behaviour. A process algebra provides operators to compose processes from simpler processes. These operators usually include an operator for sequential compositions, a choice operator and a parallel composition operator. A set of algebraic laws allow syntactic manipulations. One important law is the expansion law. It defines, how concurrency is expressed without the use of the parallel composition operator. Full-blown process algebras are Turing-complete, with all advantages and disadvantages. A verification of a design specification against the requirements specification is usually done by proving their equivalence[19]. This can either be accomplished by syntactical manipulations according to the given laws or by comparison of the semantics.

Processes can be translated to labelled transition systems which are suitable for model checkers. The history of process algebras is summarised by Baeten[18].

Petri nets[20] are also an excellent modelling formalism. A traditional Petri net (place-transition net) can be represented as a directed bipartite graph. A vertex is either a place or a transition. Input arcs connect places with transitions and output arcs connect transitions with places. Places can contain tokens and the number of tokens in the places define the state (the marking) of the Petri net. If all input places of a transition contain a token, the transition is enabled and can fire. Firing a transition consumes an input token from each input place and produces a token in each output place of the transition. If several transitions are enabled, it is non-deterministic which transition fires. A Petri net is executed by a sequence of firings. Petri nets have been used as models in many areas. Especially for concurrent systems, since they can model parallelism explicitly. A vast number of tools exist which support the design and analysis of Petri nets. There are several extensions to place-transition nets, which allow more succinct representations, improve expressiveness and allow reasoning about temporal system properties[21].

3.3 Finite State Machines

Automata theory studies abstract computing machines, especially their analysis and expressive power. A finite state automaton (FSA) is an automaton with a constant, finite amount of memory. There are deterministic FSAs (DFA) and non-deterministic FSAs (NFA) which both have the same computational power and can be converted into each other. Thus, only DFAs are considered here.

A DFA is a 5-tuple $M = (S, \Sigma, T, S_0, F)$ where S is a finite set of states, Σ is the input alphabet, $T : S \times \Sigma \rightarrow S$ is the transition function S_0 the start state and F is a set of final states.

A sequence of symbols from the input alphabet is a string. The FSA starts in the start state, reads single symbols from the string and changes its state according to the transition function. If the FSA is in a final state after the complete string is read, the string is accepted by the FSA. The set of all strings accepted by a FSA forms the language, which is recognised by the FSA. A language which can be recognised by a FSA is called regular.

Regular languages are constructively closed under union, intersection, complementation, concatenation and Kleene star. These operators can be used to compose complex state machines from simple building blocks. Furthermore, it is easy to determine whether the language recognised by a FSA is the empty set (emptiness problem) or is a subset of the language recognised by another FSA (inclusion check).

Unfortunately, FSAs also have some drawbacks. Their computational power is quite weak. It is for example not possible to model an unbounded stack, since the automata can only count a finite number of push-operations. There are some attempts

to increase the computational power of finite state machines while preserving the possibility to analyse them efficiently, for example counter-constrained finite state machines[22].

The majority of real-time systems do not terminate. The strings describing their behaviour are therefore infinite. These systems cannot be modelled with traditional finite state automata. There is another class of automata accepting infinite strings which are called ω -automata. The Büchi automaton (BA) is certainly the most popular. A Büchi automaton is also defined by a 5-tuple like the FSA above, but it has a different acceptance condition. It accepts an infinite string, if at least one final state is visited infinitely often while processing the string. Interestingly, deterministic and non-deterministic Büchi automata do not have the same computational power. The languages that are accepted by a non-deterministic Büchi automaton are the ω -regular languages. They are also constructively closed under all boolean operations and the emptiness problem is decidable[23].

But ω -regular languages are still not suitable to model real-time systems, since they only describe the order of events and not the time of their occurrence. In a seminal work by Alur and Dill[24] the timed automaton was introduced. Timed automata recognise strings of symbols which have an associated time stamp and their transitions may have timing constraints. However, if the time-stamps are real values, the languages recognised by timed automata are not closed under complementation. Using discrete time, that is the time stamps are natural numbers, preserves the closure properties[23].

There are many interrelations between the formalisms described in this section, especially between symbolic logics and automata[23]. Its beyond the scope of this survey to address them all. One of them deserves closer attention, since it is related to this project. The following section discusses problems occurring when concurrent systems are modelled with finite state machines. It concludes with a relationship between finite state machines and Petri nets which was revealed by research in this area.

3.4 FSA and concurrent systems

Finite state machines have another problem not mentioned yet. They cannot model true parallelism but only interleaving of actions. An automaton represents a set of independent (that is parallel) actions by recognising all possible linearisations. This can cause a state space explosion, since the states of the automaton are defined by the power set of the set of parallel actions. The state space explosion can seriously limit the applicability of finite state machines.

But the use of interleaving to express parallelism creates more problems. Interleaving means basically that the order of actions is determined non-deterministically. Thus, there is no way to distinguish non-deterministic choice and parallelism in the automaton model. This can cause trouble in some areas[25]. The development of

the theory of traces was partially motivated by this problem. There are three approaches to this theory, namely traces, dependence graphs and histories. All of them result in the same theory and thus only the first is considered here. A comprehensive discussion of trace theory is given by Mazurkiewicz[25].

The key idea is to define a dependency relation $D \in \Sigma \times \Sigma$ where Σ is the alphabet of the automaton. The dependence relation has to be reflexive and symmetric. If a pair of symbols is an element of the dependency relation, the symbols depend on each other and the order of their occurrence is important. Otherwise they are independent and the order is meaningless. Let us consider an example where $v = abacb$ is a string of actions and $D = \{a, c\}^2 \cup \{b, c\}^2$. Thus, the ordering of the actions a and b is meaningless and the strings in the set $\{abacb, baacb, aabcb\}$ describe the same system behaviour. In this way, the dependency relation implicitly defines an equivalence relation in Σ^* . The equivalence classes defined by this relation are called traces. A trace contains all linearisations of a behaviour with parallelism. It can be obtained by pairwise swapping of independent actions in a representative of the class. It is worth noticing that a complete dependence relation reduces all traces to singletons and thus describe common sequential system.

The dependency relation adds information about parallelism of actions to a FSA model. However, it is not really convenient to model a concurrent system with an FSA. Especially since the model quickly gets large. There are formalisms which are more suitable and create more succinct models, for example Petri nets. One interesting results of the theory of traces is that there is a class of Petri nets which have corresponding finite state machines. The elements of this class are closed under all boolean operators, concatenation, Kleene star and parallel composition. An approach described by Schmidt[7] is to conveniently model a systems as a Petri net and transform it into a FSA. The dependency relation can also be generated from the Petri net. Therefore no information gets lost, but many analysis methods developed for finite state machines can be used.

Chapter 4

Architecture Description Languages

4.1 Introduction

Architecture Description Languages (ADL) are formal languages which are designed to describe the architecture of software-intensive systems. There is some controversy about what an ADL is and what architecture exactly means[26]. Although it is not the original intention of ADLs, they tend to be used for component-based systems. The underlying idea is that software development should not focus on the writing of lines-of-code, but on a higher-level system architecture. The architecture is built from coarse-grained components and connections between the components. The idea is old and common practice in all mature engineering disciplines. First proposals to transfer this concept to software engineering were already made in the late 1960s. Unfortunately, all previous attempts to realise this vision were not satisfying. The first efforts focused on modularisation and were followed by object-oriented approaches in the 1990s. These concepts provided a basis for increasingly complex software systems. But with the exception of some specific areas (for example graphical user interfaces), building of a system completely by composition of off-the-shelf components is not possible. Although component middleware has been reasonably successful in the last years, components are often still application specific and code reuse is rare and expensive. Recent research tries to address the problems preventing software development by component composition in the past. But there are still many open questions how this can be accomplished.

Software architectures are typically described as box-and-line diagrams[5]. These diagrams consist of boxes representing some kind of components, lines between the boxes representing communication between components and some text describing the meaning of the boxes and lines. The semantics of boxes and lines are changing between diagrams or even within a diagram. Architecture description languages offer a way to express software architectures with formally defined semantics. Thus, they can improve readability and enable tools to analyse the architecture. Tool

support can include model checkers, code generators, run-time support tools and tools for the analysis of non-functional properties. This project wants to be a tiny step to provide the latter kind of tools.

Although there is little consensus what an ADL is and what it should provide, most people will agree that an ADL should at least be able to model the following explicitly[26]:

Components are entities which perform computations and can have a state. Primitive components typically represent a compilation unit. They can hierarchically be composed to composite components, which can scale up to a complete library or program. Each useful component has a *provided* interface which describes the services offered by the component. Most components also have a *required* interface specifying which services the component needs from the environment to offer its services. An ADL also has to provide means to model component interfaces. Otherwise, an analysis of the architecture is not possible.

Connectors describe the interactions between components. They do not carry out computations by itself and do not have to be compilation units, but can be implemented as shared variables, buffers, messages or any other form of communication.

Configurations represent the connections between component and connectors. They can be used to check whether the connected interfaces match. The configuration of the components and connectors finally enable an analysis of the system architecture.

Most ADLs have a textual architecture representation. Others offer a graphical notation and a few even can switch between a graphical and a textual notation seamlessly. The abilities of current ADLs vary widely. Some are simple, semi-formal notations which use annotations to describe the component and connector semantics. These kind of ADLs focus mainly on improving the understanding of the system, but offer only limited analysis capabilities. Others have a formal defined semantics and also support formal definition of component interfaces and semantics. These ADLs often provide powerful analysis tools.

One important example is Acme[27]. It was originally developed as an architectural interchange format enabling design tools to exchange architecture descriptions. Its creators recognised that each ADL has distinctive features, depending on the goals of the language designer. They wanted to combine the strength of the individual ADLs by providing a common basis. But as Acme developed, it became a simple ADL by itself. The semantics of components and interfaces in Acme is represented by property lists. They can express everything, but are not interpreted by Acme tools. The tools itself are quite limited and provide only a viewer and parser for the Acme descriptions. However, extending Acme and its use as a foundation for the development of new design and analysis tools is encouraged by the Acme Project. It does not want to be an own ADL, but rather a foundation tools can be build

upon. For this reason it represents the least common denominator of ADLs existing today[26].

On the other end of the spectrum, there is Darwin[28] which focuses on distributed and concurrent systems. It also has some support for modelling dynamic architectures by the use of dynamic instantiation. Darwin uses the Milner's π -calculus[29] to define the model's and its own semantics. π -calculus is a process algebra which considers mobile processes and dynamic configuration of communication links. Darwin comes with rich tool support including an active specification editor and a compiler to generate C++ code. The primitive components are implemented in a traditional programming language. Unfortunately, Darwin has - like most ADLs - no support for the analysis of non-functional properties.

One exception is MetaH[30], an ADL which was developed for flight control and avionics systems. Therefore MetaH provides extensive support for the development and evolution of hard real-time systems. MetaH provides hardware and operating system independent interfaces for message passing and OS-calls and has shown to be useful for the porting of real-time systems. The system model contains hardware and software components. The latter can be generated by a domain-specific code generator or implemented manually in a traditional programming language. Given the architecture and component implementation, MetaH generates the executable system. If the system designer provides execution times and path information, MetaH offers a schedulability analysis. Reliability and safety analysis are also supported. MetaH was beneficially applied in several projects and unifies many good techniques. However, it does not meet the requirements for distributed real-time systems.

4.2 Rich Architecture Description Language

The Rich Architecture Descriptions Language (RADL) [31] is an ADL which is originally derived from Darwin. RADL focuses on concurrent, distributed systems, especially systems based on component middleware like OMG's Corba, Sun's Enterprise Java Beans or Microsoft's .NET. Unlike Darwin, RADL supports the estimation of non-functional system properties and is therefore more suitable for the development of real-time systems. A diagram of a simple RADL model diagram can be found in section 5.3.

The components in RADL are named kens and usually represent the distributed components. There are basic kens which are atomic building blocks and are treated as black-boxes. The actual implementation may be unknown. But it is possible to measure and realistically model their behaviour. The composition of basic kens form composite kens. In this way kens can be assembled to ken hierarchies.

The RADL connectors are called gates. In contrast to other ADLs, gates are always implemented as gate objects. They protect the kens and control all incoming and outgoing messages. Direct communication between kens is not possible. There are two types of gates, provided and required gates. The provided gates of a ken accept

requests from other kens and specify the services the ken provides to its environment. The required gates on the other hand are services on which the ken relies on. A ken can only offer its full functionality, if the environment provides all required services. However it may be possible that a ken can provide a part of its functionality even if some requirements are not met.

A connection between gates can either be a binding or a mapping. A connection between a required gate and a provided gate is a binding. It represents communication between two kens on the same abstraction level. The control flow follows the direction of the arrow, which consists of a connection and a gate (see figure 5.3). Some connections link a provided gate with another provided gate. These are called mappings and denote communication between two abstraction levels.

4.3 RADL & Finite State Automata

Interface description languages (IDL) are used to describe the interface of components. The IDLs which are used by current middleware suffer from several problems. Most IDLs allow to specify the provided services, but not the required services. Furthermore, all common IDLs simply use signatures lists. Signature lists provide the service names and parameters, but disclose no information about the valid service call sequences. This makes components harder to use and automatic checks whether a component is used correctly are not possible[32].

RADL uses finite state automata (or an equivalent Petri net[7], see section 3.4) to specify the protocol accepted by provided gates (input protocols) and used by required gates (output protocols)[33]. The language of the FSA is the set of valid call sequences. For example, a component which provides access to a file could have a input protocol like *open, (read|write)*, close*. When a new connection between two gates is established (either static or dynamic) a compatibility check can be performed. The protocol of the origin gate has to be a sub language of the protocol provided by the destination gate. This can be done efficiently with FSAs.

The specification of input and output protocols permit to check the correctness of connections, but they are not enough to deal with non-functional properties. The prediction of the worst-case execution time of a service requires the knowledge of the exact number of service requests made during its execution. If this information is available for all services and worst-case execution times of the low-level services, which do not make service request, is known, then it is possible to calculate the worst-case execution time for the whole application by a tree-based method (see section 2.3). Unfortunately, often the number of service requests made by a specific function is not fixed, but depends on the parameters passed to the function. Since a tree-based method requires to assume the worst-case for every execution, tree-based methods can be less tight.

To predict non-functional properties, RADL makes use of dependent finite state machines (DFSM)[33]. A DFSM is a triple which consists of a set of provided protocols, a set of required protocols and the abstract machine. The abstract machine is a FSM which translates incoming requests to outgoing requests and thus effectively models a part of the component implementation. Clearly, the abstract machine has to accept all valid incoming request sequences, which are allowed according to the provided gates and may only produce outgoing requests which are accepted by the required gates. Again, this is simply a language inclusion test of regular languages and can be done efficiently by a tool. If both conditions are met, the DFSM is called consistent.

The use of DFSMs for the analysis of non-functional properties is actually an extension and not their original purpose. The concept was proposed with the intention to generalise Bertrand Meyer's Design by Contract (DbC). Meyer suggested that the use of a software entity can be regarded as a contract between the user and the used entity. A software entity might be a method or an object. If the user guarantees certain preconditions before the utilisation, then the entity guarantees that certain postconditions are met afterwards. DbC itself is a application of Hoare Logic (section 3.2) and was implemented in the object-oriented programming language Eiffel. In terms of software architecture the preconditions correspond to the required services of a component and the postconditions correspond to the provided services.

It is possible that a component can provide some services even if not all requirements are met. Reussner proposed the DFSM[34] as a generalisation of DbC. The abstract machine of a DFSM maps incoming call sequences to outgoing call sequences. If only a subset of the valid call sequences is used, then it is likely that only a true subset of outgoing call sequences is needed. The abstract machine allows to calculate this subset and to weaken the requirements of the component in a secure way. Inversely, it is also possible to calculate a new provided interface given a certain environment. The preferred direction depends on the designer and the needs of the application.

Although it was not their original purpose, DFSMs can be used successfully to predict non-functional properties. The method is in its infancy, but has been demonstrated with reliability predictions of an e-commerce application[6] and the prediction of worst-case execution times is mentioned[33]. So far, the results look promising.

Chapter 5

The system model

5.1 Robyn - the ER1 robot

This chapter proposes a RADL model of a simple mobile robot which is used in this project. The following paragraphs briefly describe the system, an ER1 robot[35] dubbed Robyn. Robyn is a mobile robot which can quickly be assembled from a hardware kit. It features a gripper, a camera, 3 IR sensors and two motors which are used to drive its wheels. The “brain” of Robyn is a computer notebook running Microsoft Windows XP and the robot control software supplied by the vendor of the robot. The robot components are connected to the notebook by USB. A second notebook computer running Microsoft Windows XP was used to control the robot remotely. The Robot Control Module (RCM) in figure 5.1 is a component which control the motors.

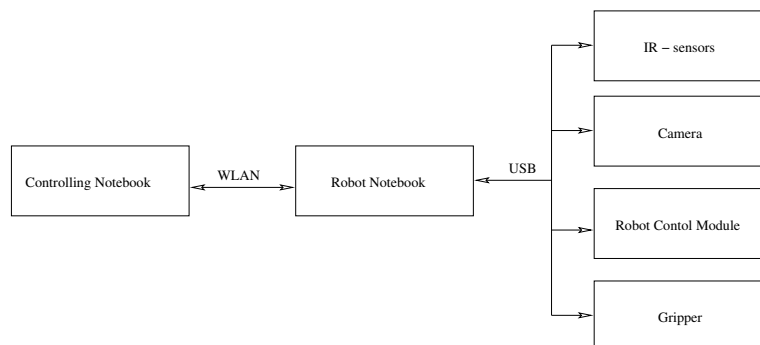


Figure 5.1: The ER1 robot system overview

The main input events of Robyn are related to vision by the camera, the IR sensors and the microphone of the notebook. Robyn can recognise objects or big areas of the same colour and supports motion detection as well as obstacle avoidance by its vision library. The sound events include an exceeding sound level and speech recognition. Robyn can react to input events by motion, playing sounds, opening

or closing the gripper and last but not least taking photos or capturing videos with its camera.

There are several programming interfaces available. The simplest one, a Windows GUI called Robot Control Centre(RCC), allows the specification of new behaviours as sequences of if-then statements. The if-conditions are the input events mentioned above, while the then-statement is one of the described reactions. Additionally, there is a “if the behaviour x has been executed” condition. It can be used to define sequences of actions. Using the Robot Control Centre GUI for programming the robot was found to be insufficient in many cases. Although the creation of simple behaviours is quickly done, more complex behaviours suffer from the limitations imposed by the GUI. Possibilities to create iterations are limited and dynamic decision which are not based on input events, for example depending on the position of the robot, are not possible at all.

Another programming interface is a Python library and its successor, the evolution robotics software platform (ERSP)[36] which contains a C++ API. The ERSP also permits running Robyn with Linux and has many additional feature like mapping, localisation and path planning. Using the ERSP frees the programmer from the limitations of the RCC GUI and allows the creation of arbitrary programs. However, ERSP is very expensive and creating new applications is a relatively complex task.

A simple but flexible interface is the socket interface. While the Robot Control Centre is running, it opens a TCP/IP socket. The robot can be controlled remotely by opening a connection to this socket and sending human readable commands over the connection. The socket interface is used in this project, since it offers enough flexibility and is still easy to program. Furthermore, controlling the robot remotely does not cost additional efforts.

The final system architecture looks like illustrated in figure 5.1. A controlling notebook is running some kind of application. The application communicates with the Robot Control Centre (RCC), which is running on the robot notebook, by a TCP/IP connection over a wireless network. The RCC offers a convenient interface to program the robot. It implements the “intelligence” of the robot, for example object recognition and speech recognition. The application can send simple requests to the RCC like “move 10 cm” or “move 90 degrees” and the RCC sends event notifications like “move done” back.

Since the application and the RCC is running under Microsoft Windows XP, this system can obviously not be a hard real-time system. Hard real-time systems either run without any operating systems or require a real-time operating systems like QNX and VxWorks. In contrast to Windows XP, these operating systems have upper bounds on critical system operations like reacting to events. The wireless network further reduces the predictability of the system. WLAN cannot guarantee maximum delivery times.

However, it was expected that the major contributions to the execution time of a behaviour is not the execution time of the software or the network connection.

These were expected to be small compared to the time consumed for the physical movement of the robot.

5.2 A sample scenario

It was decided to use a maze solving scenario for the experiment of this project. Maze solving is a well known and popular problem in robotics. It was chosen, because there are many possible variations to experiment with. Furthermore, it is not too difficult to implement and still an interesting problem. This is an important point, since this project is not writing robotics software.

Although several different maze variations were considered, only one basic experiment was actually carried out. The maze is shown in figure 5.2. It consists of 9×9 equally sized fields and was implemented with cardboard boxes.

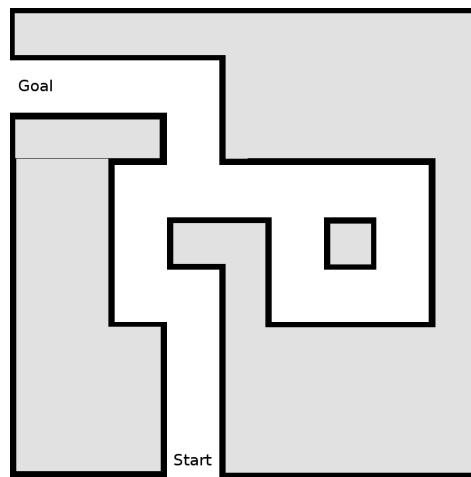


Figure 5.2: The maze used in the experiment

The right wall follower algorithm was used to solve the maze. This algorithm efficiently solves all mazes which do not have a loop around the goal. Mazes which have a start and an end point at the border fulfil this condition. The algorithm does not need additional memory, but makes decisions based on current sensor values. A human can implement the algorithm by putting his right hand on the wall to his right side. While never losing contact to the wall, he walks until the goal is found. A more formal description is given by the FSA model in figure A.3.

One reason for choosing this algorithm was that it does not need additional memory. Modelling n bits of memory with an FSA multiplies the number of states without memory by 2^n . If these states are explicitly represented, the model size explodes even for small amounts of modelled memory. There are ways to ease this problem. Firstly, most states will never be reached. It is sufficient to create a model containing the reachable states. Secondly, the states do not have to be represented explicitly. They can be represented by adding bounded variables to the FSA model. Such

an extended finite state machines can be converted to an usual FSA. However, the representation is more efficient. The FSA library implementing the tested worst-case execution time prediction method currently does not support such techniques. Since the right wall follower algorithm does not use additional memory, these techniques were also not necessary.

5.3 The model

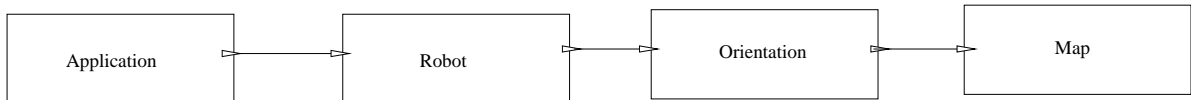


Figure 5.3: A high-level RADL model of the sample system

Figure 5.3 gives a graphical representation of the RADL model used in the experiment. The “Application” component is the maze solving program running on the controlling notebook (see figure 5.1). The “Robot” component is a composite component containing the remaining system in figure 5.1, in particular the Robot Control Centre running on the robot notebook and the USB components of the robot.

Making worst-case time predictions for the execution time of the system does not only require to model the system itself. A model of the physical environment is also needed. The environment model provides the input for the system. In the maze scenario, the robot’s input is limited to the values returned by the IR sensors. Determining these however, requires a rather complex model.

The “Orientation” component models the direction in which the robot is heading. It is also responsible for translating relative directions (for example “move forward”) of movements and measurements into absolute directions (like “move north”). Finally, the “Map” component implements the map of the maze.

For each component, an abstract machine was created which describes the semantics of the component. These models can be found in appendix A. The specification of provided and required protocols of the components was omitted. Although the protocol specifications would be part of a real RADL model, they were not required in this project.

The abstract machines were combined by the shuffle product operator. Intuitively, this operation creates an automaton that executes all abstract machines in parallel. But the abstract machines are synchronised on transitions with equal labels. Unsurprisingly, the automaton created in this case accepts only one sequence of actions. The sample scenario is deterministic and thus the resulting automaton consists of a chain of states with only one outgoing transition each. The sequence of actions

accepted by this automaton describes the progress the robot makes while solving the maze.

Chapter 6

Monitoring

6.1 Monitoring techniques

A model created by the method described in the last chapter provides the set all action sequences its system may execute. In order to predict the worst-case execution time of the system, we have to know the execution time of the atomic system operations.

This project tries to determine these execution times by measurements. It has to be verified that the execution times are independent from the execution context. If we measure different sequences of actions, the execution time of a same action should be identically distributed in all sequences. In order to verify this assumption, the system is monitored during the maze solving process. Tsai et al.[37] distinguish three kinds of monitoring approaches:

1. *Software techniques* collect the data by software and usually store the collected data in the memory of the monitored system. The instrumentation code can be inserted directly into the monitored program. This allows detailed monitoring on the program level which means that function calls and returns as well as variable value changes can be monitored. However, the monitored program has to be modified which might alter its properties. Furthermore, the instrumentation code is application specific.

A second possibility is to intercept system calls, for example by modification of the libraries which are used to make the system calls. These libraries are often linked dynamically and a recompilation of the program is therefore not necessary. This is especially useful, if the source code of the application is not available. Another advantage is that the monitoring code is application independent. But the flexibility and detail level is reduced compared to the first technique. Ordinary function calls are not visible to monitoring code.

The last technique presented here is to change the operating system kernel. In this case, the same advantages and disadvantages apply like discussed for

the last technique. Additionally, the monitoring code has access to the kernel data structures, interrupts and so on. It might be possible to implement the monitoring code more efficiently. But writing and modifying kernel code is inherently difficult and error-prone.

In general, software monitoring techniques are very flexible. Furthermore, they do not require additional hardware. A disadvantage shared by all software techniques is that they use the same system resources, in particular computation power and memory, like the monitored system. Thus, software monitoring techniques perturb the system performance. The intrusiveness of the technique depends on the amount of data which is collected and recorded. It is often necessary to find a balance between an excessive perturbation of the system performance and insufficient amount of collected data.

2. *Hardware techniques* use special devices attached to the target system's buses. They are passive and can snoop the buses without influencing the target system. The monitoring devices do not share any resources with the monitored system and the system performance remains unperturbed. However, the devices have to be able to interpret the signals on the bus and are therefore target system specific. Designing and building such devices can be complex and costly.

Another important difference between hardware and software monitoring techniques is the kind of informations which can be recorded. Since hardware monitoring devices listen on the system buses, they can only record low-level events like memory and I/O accesses. The interpretation of the recorded data requires more technical knowledge about the target system and can be rather difficult.

In summary, the main application area of hardware monitoring techniques seems to be critical hard real-time systems, where it can be worth the trouble and costs and a perturbation of the system performance is not acceptable.

3. *Hybrid techniques* try to combine the advantages of software and hardware techniques. The events are detected by monitoring code in the application or the operating system, but a special hardware device is used to record the events.

There are two common approaches how the software can trigger the recording of events. In memory-mapped monitoring, the software writes the data to be recorded on predefined memory addresses. When the monitoring device detects a write access on a predefined address, it records the data written on the address. In contrast, coprocessor monitoring uses coprocessor instructions to record events.

Hybrid monitoring techniques are as flexible as software monitoring techniques, but the perturbation of the system performance is dramatically reduced. In many cases the monitoring code can be permanently left in the monitored system. In this way, the behaviour of the system does not change by removing the code. However, if efficiency is an important issue and the monitoring code has to be removed, even the reduced influence of hybrid monitoring techniques on the system performance can be unacceptable for hard real-time systems.

The following sections discuss several possibilities to monitor the ER1 system. A software monitoring approach was found to be suitable for this project. An introduction to the instrumentation developed to carry out the measurements concludes this chapter.

6.2 Possible points of measurements

Having a look at figure 6.1 with the previous section in mind, reveals many different possibilities to monitor the ER1 systems. This section discusses a few options starting at the left side of the figure and working its way through to the right side.

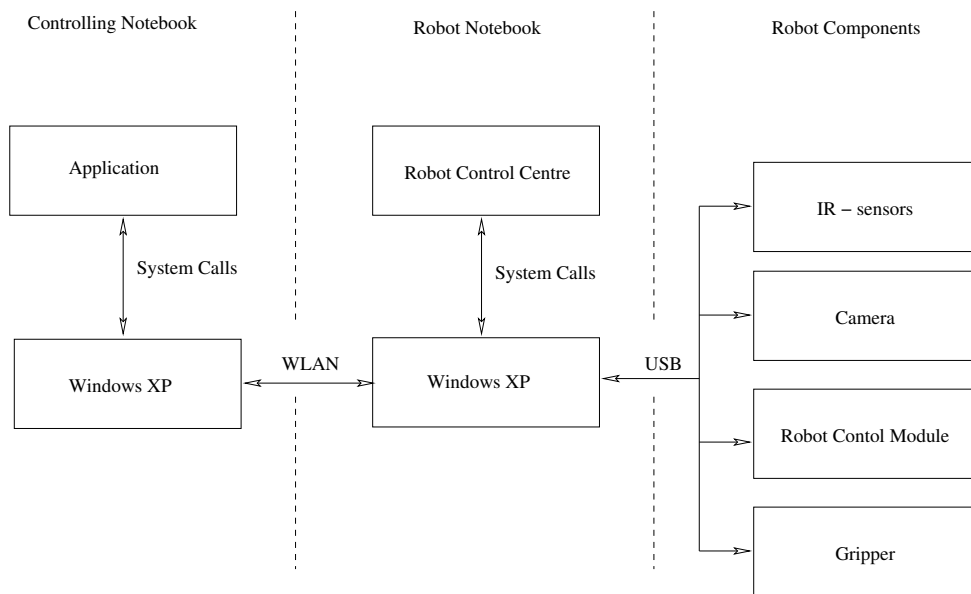


Figure 6.1: A refined system architecture diagram illustrating different monitoring options

The probably easiest way is to record events in the application itself. The source code of the application is available and there are functions to handle relevant events anyway. Monitoring code can be inserted in these functions which records the events either in a file or in the main memory. The latter option can be used to reduce the perturbation of the system performance. If the required memory is allocated in advance in the initialisation part of the application, recording of an event only required a few memory accesses. This is significantly faster than using system calls to write events into a file. An application level monitoring system is flexible and can record meaningful high-level events. The disadvantages are a slightly perturb system performance and the necessity to modify the application code.

An alternative is to detect and record events at the system call interface of the operating system or in the operating system kernel itself. Such an implementation is far more complicated. Events like system calls are on a lower abstraction level and it

is rather difficult to record useful events. Furthermore, our application is not the only process running on the notebooks. Windows XP is a multi-tasking operating system and even if the user does not start any program deliberately, a typical Windows XP installation runs over a dozen processes in the background. Therefore, many events have to be filtered and the events associated with our application have to be sorted out. Monitoring at this level does not seem to be useful for the purposes of this project.

A more promising approach is to perform network sniffing on the wireless network. That way, a third notebook can be used to detect and record events. This is comparable to a simple hardware monitoring approach with the advantages elucidated in the previous section. The monitored system remains totally unperturbed. Since the application's network protocol is human readable, the network packets can be processed easily. From the component-based system's view presented in section 5.3, the wireless network represents the connection between the application component and the robot component. The network protocol provided by the Robot Control Centre is the provided protocol of the robot component and communication over the network represents communication between the two components. Thus, the wireless network seems to be a good point to measure properties of the robot component.

On the robot notebook, monitoring can be conducted by the Robot Control Centre. We do not have access to the source code of this application, but the RCC offers to write all events concerning the robot in a log file. This functionality comes for free. However, it is unknown how the logging facility is implemented and it cannot be configured. It either records all events or it is turned off. The abstraction level of the log file is comparable to the data which can be recorded by the network sniffing approach. It is also worth to consider that the format of the log file cannot be changed. If several points of measurements are used, the file formats have to be converted to a common format which requires additional effort.

Similar to the wireless network sniffing approach, we can also eavesdrop the USB connection between the robot notebook and the robot USB components. In contrast to the wireless network packets, the packets sent over USB are harder to interpret, since the protocols are unknown and not human readable. Additionally, special hardware is required for this technique. Eavesdropping at this point appears to be useful for studying the properties of the USB components. A system model which considers the subcomponents of the composite robot component can make use of this information. But our system model is too coarse-grain for that. Therefore, no USB monitoring was carried out during this project.

6.3 Actual Implementation

Given the possibilities discussed in the previous section, we decided to study monitoring the robot via the wireless network interface. The wireless network seems to be an appropriate monitoring point, since it provides a good abstraction level and allows monitoring of the communication between the application and the robot components. However, the network sniffing approach has some problems.

Firstly, it requires a third notebook. This makes the approach rather expensive. Secondly, it can happen that a packet is successfully transferred between the controlling notebook and the robot notebook, but the monitoring notebook misses the packet. Packet loss, especially because of electromagnetic disturbances, is not unusual in wireless networks. Since the robot notebook was able to receive the packet, it will not be resend and the event is lost by the monitoring system. A third problem is that the order in which a packet is processed by the monitoring notebook and the other notebooks is not clearly defined. The monitoring notebook records the time when the packet is processed by the monitoring notebook. But the temporal distance to the time when the packet is sent and received by the other notebooks is unknown. Further monitoring, complicated by the fact that the monitoring on the three different notebooks all use different local clocks, is necessary to study this issue.

For these reasons a different approach was chosen for this project. A simple possibility to monitor the network connection is to write a program which accepts traffic on one socket and forwards the incoming packets to the actual destination. The application can now connect and send requests to the monitoring program. The monitoring program can record the events and sends the requests to the Robot Control Centre. A monitoring program based on this idea was used in the measurements described in the next chapter.

The major disadvantage of this technique is a serious performance penalty. Although both programs may run on the same computer and sending packets between them is relatively fast, the data has to traverse the TCP/IP-stack twice. Even worse, if both programs are running on the same computer, additional context switches between the programs are necessary.

Nevertheless, the technique also has a few advantages. The monitoring system is transparent to the application. If the destination host name and port can be passed as command line parameter to the application, the application does not have to be recompiled to activate or deactivate monitoring. Restarting the application with different parameters is sufficient. This flexibility can also be used by measurement scripts which can determine the run-time difference between a monitored and a unmonitored programs.

Even more important than transparency is, that this technique allows monitoring the network communication on both notebooks by running an instance of the monitoring program on the controlling notebook and another on the robot notebook. This is

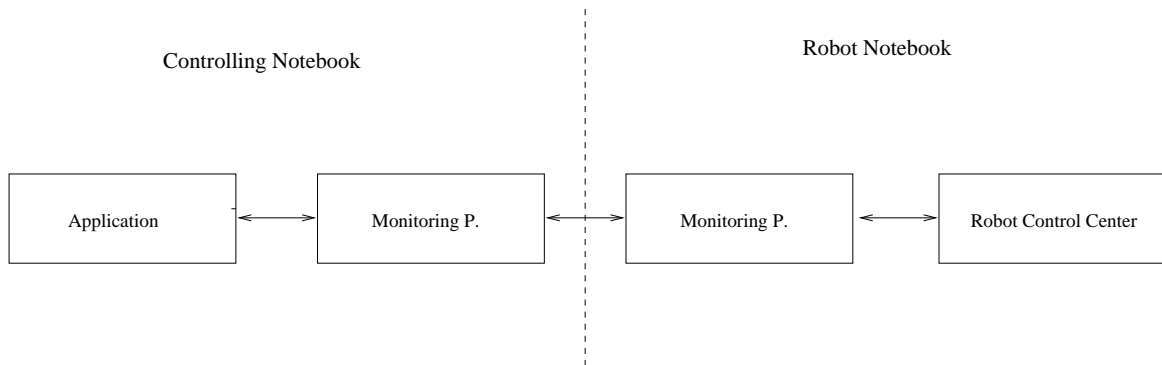


Figure 6.2: Illustration of the system monitoring approach

not possible with many other techniques presented above. The recorded data can be used to study the influence of the wireless network on the run-time. It allows to separate deviation in run-time caused by the wireless network and the deviation in run-time caused by the remaining system. Since wireless networks are immanently unreliable, this possibility seems to be useful.

Although reducing the perturbation of the target's performance by monitoring systems is an important issue in general, it has less significance in this project. The used monitoring technique is certainly expensive in terms of computational power and memory. However, the expected delay by the monitoring program was less than 1 ms and therefore less than the resolution of the system call which is used to determine the local time. The expected execution times, like moving the robot one field forward, is in the dimension of seconds. The hypothesis underlying the decision for this kind of monitoring is that monitoring overhead is negligible in this case. The ease of use and the additional data about the wireless network are considered to be more important. But, it is necessary to keep an eye on the monitoring overhead while evaluating the measurements.

Chapter 7

Measurements

7.1 Overview

The conducted measurements can be divided into three parts. Each part focuses on actions of different abstraction levels:

1. *Atomic Actions*: The first series of measurements studies the execution time of actions which are considered to be atomic for the purposes of this project. These actions correspond to the services offered by the Robot Control Centre like moving forward by 50 centimetres. Obviously, these actions are not really atomic. But the idea is to treat the Robot Control Centre as a black-box component.
2. *Cell action sequences*: This part is concerned about the possible action sequences the robot can execute in each cell of the maze. There are only 4 action sequences which the robot can execute between entering and leaving a cell. These action sequences consist of a small number of atomic actions (between 3 and 7).
3. *Maze*: The last part examines the execution time of solving a whole maze. The maze used for this series of measurements is already shown in section 5.2. The maze can either be considered as a sequence of atomic actions or as a sequence of cell action sequences.

The driving force behind the layout is the desire to predict non-functional system properties by compositional reasoning. It should be possible to predict the execution time of a complex action by measuring the execution times of its atomic actions. This choice of measurements allows studying compositional reasoning about execution times of the ER1 robot. The measured execution times for atomic operations are used to concretise the system specification. They turn the abstract system specification to a concrete system implementation which is used as input for the worst-case execution time predictions in the next chapter.

7.2 Exploratory Data Analysis

Exploratory Data Analysis(EDA)[38] is used to examine the measured data. EDA is a data analysis approach which relies on graphical analysis techniques. The atomic action measurements are analysed by a 4-plot. This is a collection of plots that can be used to check common assumptions about the measured data. The 4-plot contains a

1. *run sequence plot*: A run sequence plot illustrates the measured values versus the number of the measurement. It can be used to verify that the data of a series comes from a fixed distribution with a fixed location and variation.
2. *lag plot*: A lag plot is used to check the randomness of the data. If the lag plot shows a non-random pattern, this is a strong sign for autocorrelation. A lag of one is used in all lag plots.
3. *histogram*: The histogram gives an impression of the distribution from which the measured data comes. All histograms in this section also contain a corresponding density function which is generated by kernel smoothing (intuitively, kernel smoothing creates a density function by calculating a kind of weighted average).
4. *normal probability plot*: The normal probability plot reveals, whether the data is approximately normally distributed. Since the normal distribution has many nice properties, this is often an interesting question. It is not too important for this project, but the plot can reveal some additional information.

Some other supportive plots, like a comparison of several series of measurements and autocorrelation plots, are also used where they seem to be appropriate. Unfortunately, it is not possible to analyse every gory detail of data series within the scope of this thesis. The point of the analysis in the following sections is to check some important assumptions (like randomness of the measurement series) and to provide a basic understanding of the ER1 system's temporal behaviour.

7.3 Atomic actions

The execution times in all measurement series were measured by the application which controlled the robot during the measurements. Thus, the execution times include network delays, the computation time on the robot notebook and the time for the physical execution. The measurements were conducted twice, one time with the monitoring system introduced in section 6.3 on both notebooks and the other time without the monitoring system. The next sections describe the measurements without monitoring. Section 7.6 compare the results with and without monitoring.

There are three atomic robot actions that occur in the maze scenario: move forward, turn (left or right) and read IR sensor. The following sections briefly describe how the execution times were measured and summarise the results.

7.3.1 Move forward

The forward movement execution times were measured 60 times. The measurements were taken in two series with 30 measurements each. The two series were compared with a Wilcoxon test, which is a non-parametric alternative for a paired t-test. The Wilcoxon test revealed a p-value of 0.2579. It is therefore assumed that all data samples are from the same population. This section examines the concatenated data set. Consequently, the following lag plot is slightly disturbed. A separate autocorrelation plot for each series is provided in appendix B.

A quantitative summary of the data in milliseconds given by the following table:

Mean	S.D.	95% C.I.(Mean)	Minimum	Median	Maximum
6443	90	(6420,6466)	6274	6454	6604

Table 7.1: Summary of the forward movement execution time measurements

The run sequence diagram shows no significant shift in location or variation. This confirms the fixed distribution assumption. The lag plot looks random and does not show any autocorrelation. The result is supported by the autocorrelation plots.

The histogram suggests that the measurements come from a mixture of 2 approximately normal populations. The data is clearly not normal distributed which is also confirmed by the normal probability plot.

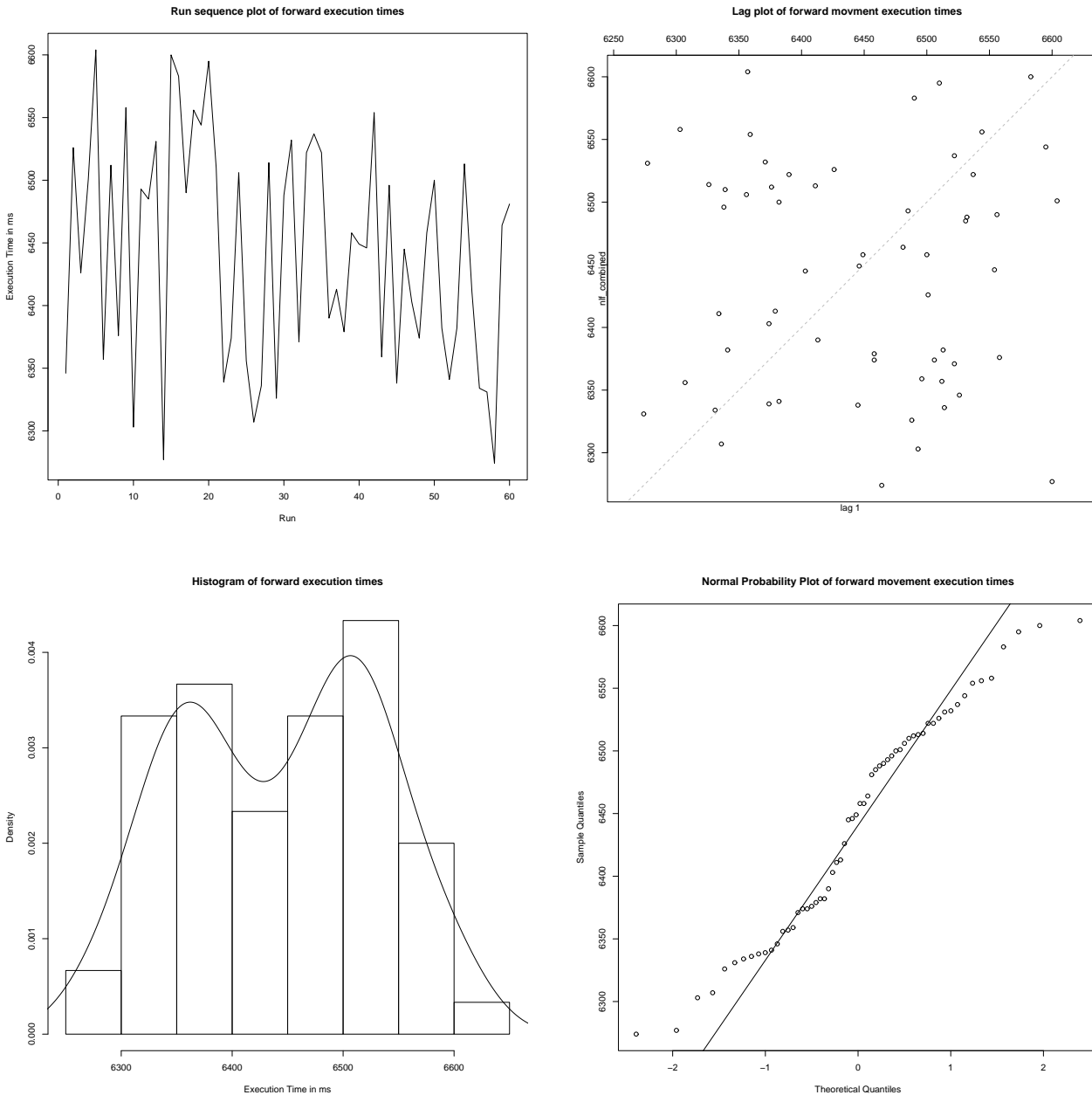


Figure 7.1: 4-plot of the forward movement execution time without monitoring

7.3.2 Turns

The measurement of turn execution times were carried out twice. One series measured the 90° left turn execution times and the other 90° right turn execution times. Each series consists of 48 measurements. There is no good reason to believe that the two turns have different execution times. The Mann-Whitney test is a non-parametric test which can be used to check whether two independent groups of sampled data come from the same population. Unlike a unpaired t-test, the Mann-Whitney test does not assume a particular distribution. Using a Mann-Whitney test for comparison of the left and right turn execution times reveals a p-value of 0.212. Therefore, we assume that both series come from the same population and examine a concatenated data set with 96 values. Again, this has an objectionable influence on the lag plot. Separate autocorrelation plots and a graphical comparison of the sample histograms are provided in appendix B.

A quantitative summary of the data is given in milliseconds by the following table:

Mean	S.D.	95% C.I.(Mean)	Minimum	Median	Maximum
7139	113	(7116,7161)	6938	7147	7682

Table 7.2: Summary of the turn execution time measurements

Again, the run sequence chart reveals no shift in location or variance of the distribution. The left and the right of half of the diagram look similar. This coincides with our assumption that left and right turn execution times are equally distributed. The chart points out one outlier: the 68th measurement takes significantly longer.

The lag plot and the autocorrelation plots support the claim that the data is random and not autocorrelated. Interestingly, the histogram looks like a mixture of 2 populations. Although the two populations seems to be less separated, the distance between the means of these populations is similar to the distance between the means of the two forward movement populations. Perhaps, the same mechanism is involved.

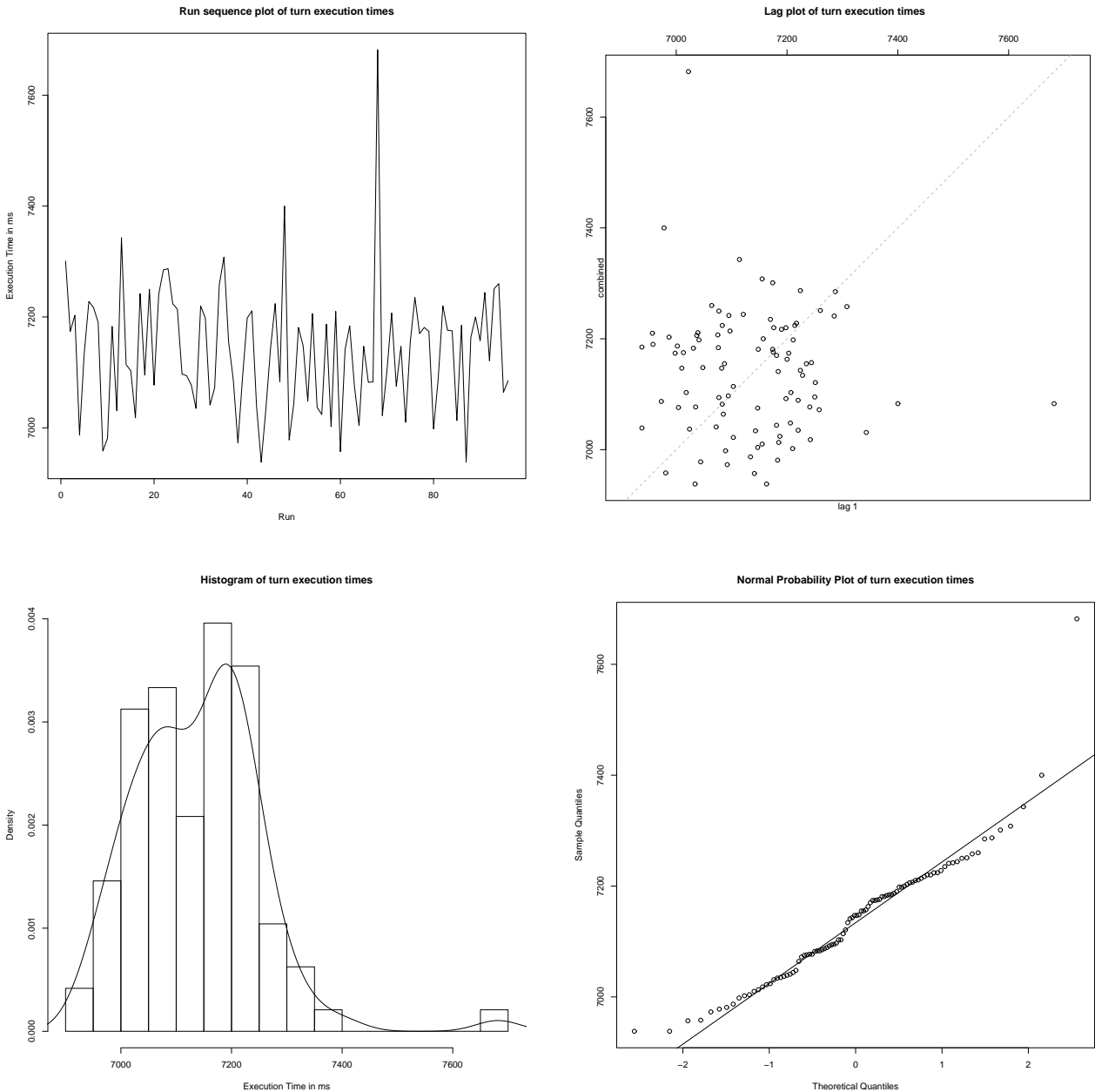


Figure 7.2: 4-plot of the turn execution time without monitoring

7.3.3 IR sensors

Many different series measuring the IR sensor reading times were carried out. All measurement series consist of 200 values. Instead of giving all details, only the results are summarised and some important issues are discussed in depth.

During the analysis of the measurement series, a few important theorems were verified:

1. The two different sensors used in the maze scenario (forward and right sensor) have the same distribution of sensor reading times.
2. The execution time does not depend on the value which is read by the sensor.
3. If there is a break of more than 1 second between two sensor readings, the measured execution times are not autocorrelated and come from a fixed distribution.

Unfortunately, the IR sensor reading times are not always independent from each other. If a sensor is read many times within a short period of time, the reading time increases dramatically. This is illustrated by figure 7.3. With a break of more than one second between two reading requests, this behaviour disappears. A 4-plot and an autocorrelation plot for a series with a 1 second break between two readings is given in appendix B.

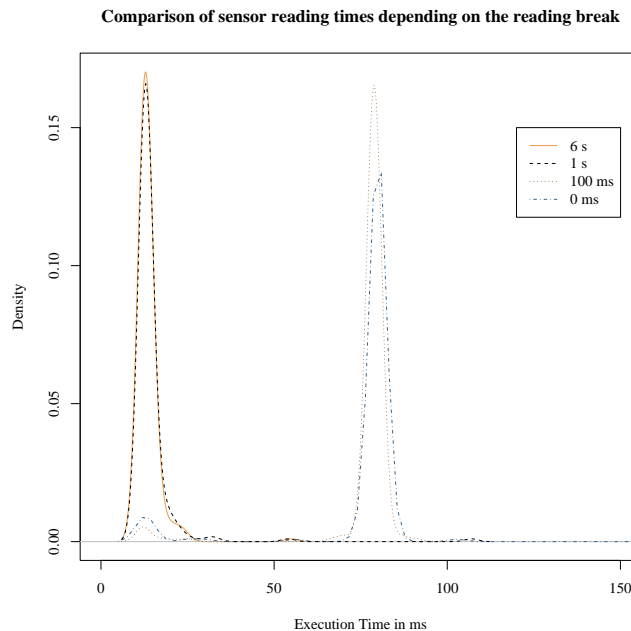


Figure 7.3: Comparison of the IR sensor reading times depending on the break between two readings

The IR sensor reading action is different from the other actions. The execution times are very short and influences on the execution time which are too small to be visible for other actions, are significant for the IR sensor reading times. This includes delays by the wireless network and monitoring.

The situation is amplified by the fact that one sensor reading request by the application is translated to five repetitive sensor readings by the library implementing the network communication. The values returned by the IR sensors are relatively unreliable. Therefore, the library repeats the measurement and calculates the average of the measurements. Unless something else is stated, a sensor reading in this thesis always refers to 5 actual readings. The finite state machine model also abstracts from this fact.

The following tables summarises the data of the measurement series with 1 second breaks in milliseconds:

Mean	S.D.	95% C.I.(Mean)	Minimum	Median	Maximum
14	8	(13,15)	12	13	107

Table 7.3: Summary of the IR sensor reading time measurements

Further experiments confirmed that the wireless network has an important influence on IR sensor reading times and the high peaks of the execution time are due to the wireless network. A series of measurements was carried out with the monitoring program running on each notebook. A 6 seconds break was made between the readings. A series of 200 sensor readings by the application results in 2000 packets send over the network (that is a request and a reply for each of the 5 actual readings).

Determining the time a packet needs to transit the wireless network is not easy, because the two monitoring programs use different local clocks. However, we can calculate the packet round trip time. The time between the request and reply packet measured by the monitoring program on the robot notebook is subtracted from the time between the request and reply packet on the controlling notebook. The result is the time which the request and reply packets together needed to transit the wireless network. Since only local time differences are used in this calculation, the difference between the local clocks is of no relevance.

The packet round trip time determined by this method is illustrated in the packet round trip time figure in appendix B. The highest round trip time measured is 391 ms and the mean value is 3.882 ms. In contrast, a single reading of the sensor by the Robot Control Centre on the robot notebook typically took between 1 ms and 2 ms, with a mean value of 1.34 ms. None of the outliers was greater than 16 ms.

7.4 Cell Sequences

One view of the maze solving process performed by the robot looks at the process as a sequence of the atomic operations studied in the previous section. On a higher abstraction level, the robot processes a sequence of different cells. Looking at the right wall follower algorithm FSA in appendix A reveals 4 different sequences which can be executed between entering and leaving a cell:

1. sequence:
 $!ir_right \rightarrow \sim ir_right=free \rightarrow !turn_right \rightarrow \sim turn_right \rightarrow !move_forward \rightarrow \sim move_forward$
2. sequence:
 $!ir_right \rightarrow \sim ir_right=blocked \rightarrow !ir_forward \rightarrow \sim ir_forward=free \rightarrow !move_forward \rightarrow \sim move_forward$
3. sequence:
 $!ir_right \rightarrow \sim ir_right=blocked \rightarrow !ir_forward \rightarrow \sim ir_forward=blocked \rightarrow !turn_left \rightarrow \sim turn_left \rightarrow !ir_forward \rightarrow \sim ir_forward=free \rightarrow !move_forward \rightarrow \sim move_forward$
4. sequence:
 $!ir_right \rightarrow \sim ir_right=blocked \rightarrow !ir_forward \rightarrow \sim ir_forward=blocked \rightarrow !turn_left \rightarrow \sim turn_left \rightarrow !ir_forward \rightarrow \sim ir_forward=blocked \rightarrow !turn_left \rightarrow \sim turn_left \rightarrow !ir_forward \rightarrow \sim ir_forward=free \rightarrow !move_forward \rightarrow \sim move_forward$

The FSA itself actually allows an infinite number of different sequences. The loop on the left side is unbounded and can be repeated infinitely often. In a proper static maze however, the robot never turns left more than twice, before the field in front of the robot is free. The loop bound is therefore not determined by the algorithm but by the environment of the robot.

A measurement series with 30 executions was carried out for each series. In all cases, the monitoring program was running on both notebooks. The following table summarises the results in milliseconds:

Sequence	Mean(ms)	S.D.	95% C.I.(Mean)	Minimum	Median	Maximum
1	13566	127	(13518,13612)	13259	13590	13754
2	6553	77	(6524,6582)	6323	6578	6650
3	13732	111	(13731,13773)	13477	13754	13907
4	20948	159	(20889,21008)	20653	20946	21389

Table 7.4: Summary of the sequence execution time measurements

An partial analysis of the data is presented in the chapter 8.

7.5 The Maze

Finally, the maze solving execution times were measured. The used maze is described in section 5.2. The process was measured 5 times with monitoring on both notebooks and 5 times without any monitoring. The number of measurements is very low. However, the execution times are long and the number of measurements in one series is limited due to the battery power of the robot. At the same time, the results seem to be definite. The following table shows the results in seconds:

Monitoring	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run 5 (s)	Mean	S. D.
Yes	222.02	221.72	222.23	221.90	221.64	221.90	0.24
No	219.29	218.94	219.19	219.40	219.80	219.32	0.32

Table 7.5: The maze solving time measurements

A more comprehensive analysis of the data follows together with the predicted maze solving times in the chapter 8.

7.6 Monitoring influence on system performance

The data in the previous section indicates a significant monitoring influence on the system performance. The run time difference in the maze solving measurements is about 1.2%. This section takes a closer look at the reasons of this difference.

As already mentioned, the relative monitoring influence on forward movements and turns are expected to be negligible. The delay by the monitoring program is assumed to be depending on the number of packets required for a certain action, but independent of kind of action.

The IR sensor readings seem susceptible to monitoring influences, since the execution times of the IR sensor readings are relatively short. The relative influence increases when the execution time of the action decreases. The following table compares the sensor reading times with and without monitoring in milliseconds:

Monitoring	Mean	S.D.	95% C.I.(Mean)	Minimum	Median	Maximum
Yes	36	28	(32,40)	28	34	398
No	14	8	(13,15)	12	13	107

Table 7.6: Comparison of IR sensor reading times with and without monitoring

The higher standard deviation of the measurements with monitoring is mainly caused by two outliers. More interesting is the difference of the mean values. The monitored IR sensor readings take 157% longer in average. This is much more than expected. Remembering that each IR sensor reading results 5 actual sensor readings and therefore 10 network packets (request and reply each), we can estimated

an average monitoring delay per network packet of $(36ms - 14ms)/10 = 2.2ms$. Since there is one monitoring program on each notebook, the delay caused by one program is 1.1 ms. A delay of this order of magnitude was expected (see 6.3). But the impact of this an delay was underestimated.

A surprising effect is uncovered by comparing the monitored and unmonitored measurement results of the forward movements and turns. The difference of the forward movement mean values (turn mean values) is 64 ms (85 ms). A forward movement or turn requires 7 network packets. Therefore, a much smaller delay was expected. The cause for this high delay remains unclear. However, a graphical comparison (see appendix B) provides a hint. In the unmonitored case, the density functions seems to be a mixture of two populations. The monitoring causes the population with the lesser mean value to disappear.

The monitoring overhead is larger than expected. The hypothesis that the overhead can be neglected is falsified. Although the packet sizes differ only by a few bytes, the monitoring overhead is not independent from the action. Compensating the monitoring influence on the measured values is therefore complicated.

The prediction and evaluation in the next chapter has to pay attention to the monitoring influence. All measurement series except the sequence executions were done with and without monitoring. As long as the differently acquired data is not mixed in a calculation or graph, the monitoring delay is unproblematic. If monitoring influence would have been a critical issue in this project, another monitoring approach would have been chosen (see 6.3).

Chapter 8

Worst-case Execution Time Predictions

8.1 Algorithm

The worst-case execution time predictions in this chapter are based on a simple algorithm. This algorithm takes the FSA model of the system (see section 5.3) and the execution times determined in the last chapter as input.

The algorithm assumes that the FSA is acyclic. This is no severe restriction. If a FSA has cycles with an upper bound on the number of repetitions, the FSA can be made acyclic by loop unrolling. If no upper bound exists, the system can loop forever and a worst-case execution time does not exist.

One example for a loop with an upper bound is the application FSA implementing the right wall following algorithm in appendix A. The loop in the left side is limited to 2 repetitions by the environment. The argument is that after two turns the robot will face to the direction it was coming from. Thus, the field is always free. Using 2 as upper bound of the loop, the worst-case execution time for processing one field of the maze can be calculated.

Given a system FSA and execution times for each symbol, determining the worst-case execution time is equivalent to finding the longest path in a weighted graph. Although this problem is NP-complete in general, it can be solved efficiently in directed acyclic graphs. The path begins at the initial state S_0 and ends at the final state S_f . If the system FSA has several final states, a new final state is created and transitions with weight 0 from all former final states to the new state are added.

Let (S, Σ, T, S_0, S_f) be a FSA model of a system with only one final state S_f . $(S_1, S_2, A) \in T$ means that the FSA makes a transition from state S_1 to S_2 by action A . Furthermore, $W : \Sigma \rightarrow \mathbb{N}$ define the execution time for each input symbol

and $I(S)$ is the set of all incoming transitions to state S . The function $WCET(S)$ returns length of the longest path from S_0 to state S .

$WCET(S)$ can be calculated recursively:

- $WCET(S_0) = 0$
- $WCET(S) = \max_{(S_i, S, A) \in I(S)} \{WCET(S_i) + W(A)\}$

The worst-case execution time of the system is defined as $WCET(S_f)$.

8.2 Prediction preconditions

Predicting worst-case execution times using the method described in the previous section works well, if certain conditions are met. Firstly, the system model has to be accurate enough to make reasonable predictions. In particular, the number of outgoing requests made by the components should be relatively accurate. This condition can easily be fulfilled for the system used in this project. However, if more complex algorithms are involved, finite state machines can quickly get complicated. More powerful notations which can be translated to finite state machines could ease this problem. Some formalisms presented in chapter 3 can be used for this purpose.

A second condition is that the executions times of the single actions are independent from each other. Since our system is sequential and the measurement series do

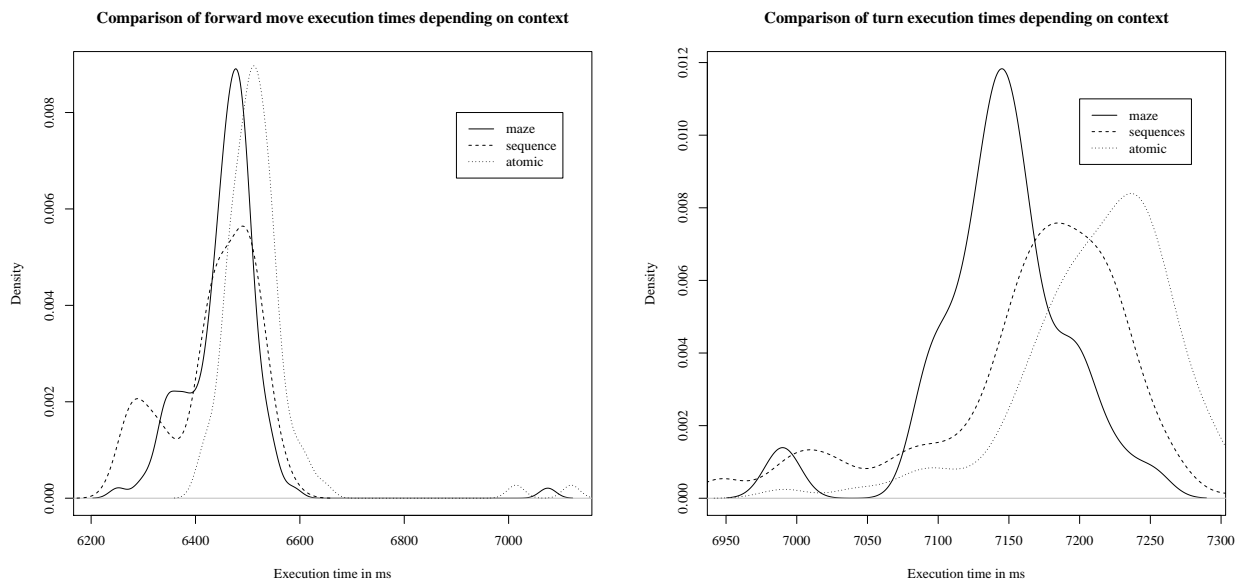


Figure 8.1: Execution times of forward movements and turns depending on execution context

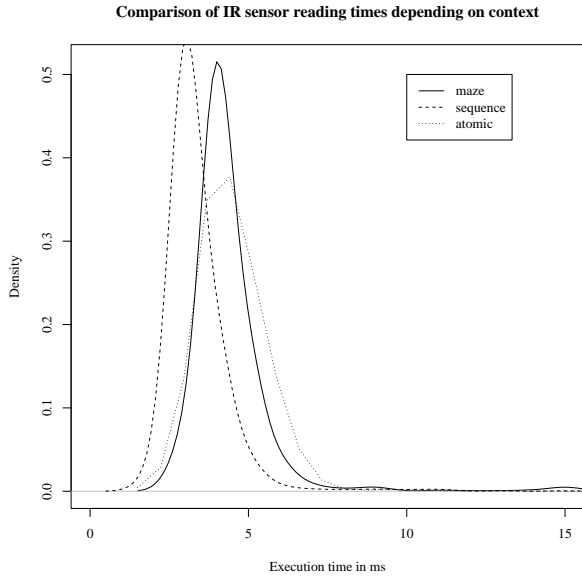


Figure 8.2: IR sensor reading times depending on execution context

not show any autocorrelation, this system might fulfil the independence condition. For distributed real-time systems in general, this is rather uncommon. In most distributed systems, different actions compete for the shared resources like processors. Such dependencies can be modelled as finite state machines by adding additional synchronisation. But it usually requires a diligent analysis and the model will get far more complex than the relative simple RADL model presented here.

For the purpose of this project, we try to verify the independence assumption with the help of the monitoring system. The plots in figure 8.1 and 8.2 compare the distribution of execution times measured in isolation to the distribution of execution times measured during the sequence measurements and the maze solving process. All series illustrated in these plots were measured with monitoring and logged by the monitoring program on the controlling notebook.

At least, the corresponding density functions look similar. But from a statistical point of view, they are all different from each other. For example, a Mann-Whitney test of the atomic and maze forward movements reveals a p-value of 1.144E-12. The two groups of measurements are therefore likely to come from two different populations. The following table shows the different mean values in milliseconds:

Action	Atomic (unm. ¹)	Atomic (m. ¹)	Sequences	Maze
Forward Movement	6443	6522	6437	6457
Turn	7139	7218	7169	7144
IR sensor	2.9	5.0	5.4	6.7

Table 8.1: Sample means of action execution times depending on context

¹unm. = unmonitored m. = monitored

The IR sensor reading time refers to one actual reading, not a composite IR sensor reading operation. Although there are 1000 measurements (a statistically significant number) in each group, their mean values are quite different. In comparison with the diagram, the mean values might surprise. The discrepancy is caused by outliers. The number of outliers is mainly influenced by the reliability of the wireless network which depends on the intensity of electromagnetic disturbances and the distance between the two notebooks. Both variables may change over time. Thus, greater mean value of the maze measurements might be caused by a greater distance between the notebooks during the measurements.

It is interesting to note, that the unexpected monitoring overhead of the forward movements and turns is reduced in the sequence and maze measurements. During the maze solving process, the mean values of the atomic actions are almost equal to the mean values of the unmonitored atomic actions. This is another violation of the independence assumption. Since the monitored isolated actions take longer than the execution in the maze, the predictions based on the monitored execution times will overestimate the maze solving time. However, there is no fundamental reason, why it could be the other way round. The independence assumption therefore always needs to be checked carefully.

In summary, the execution times of our atomic operations are not context independent. A refined, more detailed model with low-level atomic action might be able to represent the dependencies between the actions. These low-level actions could be truly independent. However, the wireless network would still be a problem. The nature of the system makes very accurate predictions difficult. It is possible to make estimation which abstract from some details like the wireless network's influence. In this case, the execution times should be measured in a similar environment like the environment expected during the later application of the system. For example, the level of electromagnetic disturbances should be similar.

In the predictions made in this chapter, we assume that the execution times are independent from the context. There are indications that the dependencies of the monitored actions are related to the monitoring overhead. Thus, it is assumed that the violations of the independent assumption is less severe in the unmonitored case. The predictions using the monitored atomic actions will significantly overestimate the real execution time, since the system performed better in the sequences and mazes than in the atomic measurements.

8.3 Soft worst-case execution times

The prediction method described in section 8.1 requires an associated fixed value with each symbol. Schmidt et al.[7] used this method to predict the worst-case execution time of a robotic system. They associated the worst-case execution time of each action with the symbols.

This is not possible for the ER1 system, because there are no finite execution time bounds. The wireless network could theoretically take an arbitrary long time to deliver a packet, although the probability that a packet needs more than a second is quite low. The ER1 system can only be a soft real-time system and the predicted worst-case execution times are only execution times which are rarely exceeded. Keeping this in mind, there are several reasonable values that could be associated with the symbols:

- *worst-case measurement*: Using the longest measured execution time is a self suggesting possibility. While being aware of the fact that this time could be exceeded with a small possibility, this will rarely happen. Thus, assuming the longest measured execution time is the worst-case execution time should provide safe predictions. This choice has one major drawback. If the longest measured execution time is multiple times greater than the average execution time, the predictions may be safe but uselessly long. An example is the IR sensor reading time with a mean value of 14 ms, but a longest measured execution time of several hundred milliseconds. It is worth noting that the longest measurement is a variable which monotonically increases when the number of measurements increases. It is always possible to measure a longer value and therefore increase the longest measurement, but a large number of smaller measurements does not decrease it again.
- *0.95-percentiles*: The problems of taking the longest measurement can be eased by using the 0.95-percentile of the execution time. The idea is to drop outliers and get a more reasonable prediction. A single action might eventually exceed the time associated with its symbol, but many others will stay well below and compensate for the excess. This rule can be generalised by choosing a threshold t (here: $t=5\%$). The threshold can be adapted depending on the system and the required conservativeness of the predictions.
- *mean time + 2* σ* : A totally other approach is based on the measured average times ($\mu_{forward}, \mu_{turn}, \mu_{ir_sensor}$) and standard deviations ($\sigma_{forward}, \sigma_{turn}, \sigma_{ir_sensor}$). In order to get more accurate predictions, it might be useful to predict the mean execution time and estimate the execution time variance. If the action execution times are independent from the context, it should be possible to predict the average execution time of the system very accurately. According to the central limit theorem, the distribution of system execution time will converge to a normal distribution as the action sequences of a system get longer. Thus, the distribution of the system execution time can be approximated by a normal distribution with the predicted mean value and variance. The parameters of the normal distribution are estimated by:

$$\mu = \#forward * \mu_{forward} + \#turn * \mu_{turn} + \#ir_sensor * \mu_{ir_sensor}$$

$$\sigma^2 = \#forward * \sigma_{forward}^2 + \#turn * \sigma_{turn}^2 + \#ir_sensor * \sigma_{ir_sensor}^2$$

Finally, the estimated distribution of the system execution time is used to calculate a worst-case execution time. In this chapter, we add $2 * \sigma$ to the mean execution time. The resulting execution time is supposed to hold in 97.7% of system executions.

The last approach has a serious problem. It works well with the given sample scenario, since it is deterministic. However, in a non-deterministic system the prediction algorithm has to compare partial executions while searching the longest execution path. It is not possible to define an order on partial system executions which guarantees that the path with the greatest $\mu + 2 * \sigma$ value is found. Either a heuristic has to be used with the risk that the predictions are not safe, or another algorithm like examining all possible sequences have to be used. The latter possibility is safe, but for a large number of possible sequences, the run-time for making the prediction is prohibitive.

8.4 Predictions

8.4.1 Parameters

Predictions with all three possibilities described in the previous section are presented. Two different sets of parameters are used to make the predictions. The first set P_1 is taken from the atomic action measurement series without monitoring presented in the last chapter. The next tables shows these parameters in milliseconds:

Method	Forward Movement	Turns	IR sensor reading
Longest Measurement	6604	7682	107
0.95-Percentile	6558	7301	19
Mean+2 * σ (Mean/S.D.)	6443/90	7139/113	14/8

Table 8.2: Prediction parameters without monitoring

The other set of parameters, P_2 , is used for predictions with monitoring. The values were measured by the application conducting the measurement of atomic actions with monitoring on both notebooks. Therefore, the values contain all network communication and computations. The following table shows P_2 in milliseconds:

Method	Forward Movement	Turns	IR sensor reading
Longest Measurement	7124	7719	398
0.95-Percentile	6613	7294	37
Mean+2 * σ (Mean/S.D)	6527/94	7224/75	36/28

Table 8.3: Prediction parameters with monitoring

Predictions based on both parameter sets neglect the computation time consumed by the application on the controlling notebook between the calls to the robot control

library. A short look at the application confirms that no complex or time consuming operations are carried out by the application in the neglected parts of the program. The run-time is assumed to be evanescent.

8.4.2 Sequences

The cell sequences were only measured with monitoring. Thus, only predictions with the P_2 parameters were made. The last column shows the longest measured sequence execution time:

Sequence	Longest M.	0.95-P.	Mean+2 * σ	Measurement(worst)
1	15.24 +10.8%	13.94 +1.4%	14.04 +2.1%	13.75 0.0%
2	7.92 +19.1%	6.69 +0.6%	6.80 +2.3%	6.65 0.0%
3	16.04 +15.3%	14.02 +0.8%	14.12 +1.5%	13.91 0.0%
4	24.15 +12.9%	21.35 -0.2%	21.42 +0.1%	21.39 0%

Table 8.4: Monitored sequence predictions

As expected, the taking the longest measurement as execution time is very conservative and results in significant overestimations even for small sequences. A part of this overestimation is due to the context dependencies uncovered in section 8.2. However, these dependencies are not severe enough to justify overestimations of 10%-20%. The other two methods produce relative tight results. For short sequences, Mean+2 * σ seems to be more conservative than the 0.95-Percentile.

The underestimation made by the 0.95-Percentile method is awkward. But the three longest of 30 sequence 4 measurements are 21.39 s, 21.20 s and 21.16 s. In this case, the longest sequence measurement appears to be exceptionally long. It was already mentioned that there is no really secure worst-case execution time for this system. The estimate holds for all 29 other executions and is therefore considered to be satisfactory.

A problem for the mean+2* σ method is the small number of actions in the sequences. Since the method is based on the central limit theorem, the method requires relatively large execution sequences and should not be used in this case. Unfortunately, there is no general rule how long the execution sequence have to be. The required length is depending on the execution time distributions of the atomic actions. The more these distributions differ from normal distributions, the longer is should the sequences be. The maze execution sequences are long enough for this method.

8.4.3 Maze

The maze executions were measured with and without monitoring. The unmonitored maze executions are predicted using parameters P_1 and monitored maze executions using parameter P_2 . A Java program implementing the FSA model introduced in section 5.2 and an implementation of the prediction algorithm was used to make the predictions. Only the standard deviation for the mean+2* σ method was calculated and added manually. The calculation was based on the predicted action sequence.

The following table shows the predictions and compares them to the longest measured execution time.

	Longest M.	0.95-P.	Mean+2 * σ	Measurement(worst)
unmonitored	232.88 +6.0%	224.03 +1.9%	220.69 +0.4%	219.80 0.0%
monitored	259.95 +16.97%	226.17 +1.78%	224.53 +1.0%	222.23 0.0%

Table 8.5: Maze worst-case execution time predictions in seconds

These predictions primarily confirm the results of the sequence predictions. While the longest measurement method seriously overestimates the execution time, the other two methods produce comparatively tight predictions. None of the predictions were exceeded.

In the sequence prediction case, the 0.95-percentile method is more conservative than mean+2 * σ . Whereas the situation is reversed in the maze prediction case. The reason is that the maze execution sequences are much longer. With an increasing number of actions, using the 0.95-percentile parameters makes the estimation drifting away from the mean execution time linearly. In contrast, using the mean+2 * σ method results in an estimation drifting away from the mean execution time proportionally to the square root of the number of actions. Thus, the mean+2 * σ method is supposed to produce tighter results with long sequences. Additionally, long sequences also justify the use of the central limit theorem.

The maze predictions allow to compare the monitored and unmonitored case. The overestimation of the monitored predictions is expected due to the dependency issues discussed in section 8.2. The unmonitored predictions are tighter and support the assumption that the dependencies are mainly caused by the monitoring system. The unmonitored system behaves better and the independence assumption seem to be more appropriate. This is also supported by mean execution time predictions made for the mean+2 * σ predictions. The predicted mean execution time for the unmonitored system is 219.57 seconds, only 0.1% more the the measured mean execution time (219.32 s). In contrast, the predicted monitored mean time is 223.43 s and the measured mean execution time is 221.9 s.

Chapter 9

Results & Conclusions

9.1 Results

The initial objective of this project was to study and verify a method to make worst-case execution time predictions based on a RADL model. In particular, the following research questions have been examined:

1. Can a RADL model be used to predict the worst-case execution time of a real system?

Yes, for systems like the ER1 robot it is even possible to make reasonable predictions with a coarse model. However, RADL models which are suitable for execution time predictions have to be far more complex than models which are only used to check the compatibility of the system components.

It was also shown that choosing the right abstraction level is critical. The abstraction level has to be low enough to allow modelling of dependencies between the atomic system operations.

2. Do measurements of the real system confirm the predicted worst-case execution times?

In nearly all cases yes, but one prediction was exceeded by a measurement. However, this measurement was significantly longer than all other values in its measurement series. We learnt that a real worst-case execution time does not exist for systems like the ER1 robot. Neither the used operating system (Windows XP) nor the wireless network can give execution time guarantees. For such systems, it is only possible to determine execution times which are rarely exceeded, but not guaranteed. Thus, a single measurement which exceeds the estimation is not unexpected and does not invalidate the prediction method.

3. How tight are the predictions? Are the predictions too pessimistic to be useful?

The answer to this questions depends on the chosen execution times for the atomic operations. Three different choices have been proposed and tested:

- *Longest measurement*: Taking the longest measured execution time of each atomic operation was found to cause a serious overestimation of the system execution time. The estimations were between 6%-20% above the longest measured system execution.
- *0.95-Percentile*: This method assigns an execution time to each atomic operation which is not exceeded in more than 95% of all measurements of this atomic operation. The estimation was always within 2% of the longest measured execution time. The 95%-percentile rule seems to be a good choice. However, the only prediction which was exceeded by a measurement is also made with this method.
- *Mean+2*σ*: This last method actually does not work well together with the used prediction algorithm. It was tested nevertheless, since it was expected to produce the most accurate results for long executions. This hypothesis was confirmed. The maze execution time predictions were both within 1% of the longest measured execution. Unfortunately, this method requires a less efficient prediction algorithm for non-deterministic systems.

In summary, the 0.95-percentile method can be used to make reasonable predictions and is the preferred choice for most systems. The 5% threshold might have to be adapted to the specific system. For example, a 0.99-percentile can sometimes be more appropriate.

Another finding is that adding small components like the monitoring program to the system can have an unexpected high influence on the system performance. While the monitoring program was not designed to minimise the influence on the system, the magnitude of the influence was underestimated. Especially surprising was, that the monitoring overhead was not predictable and the monitoring increased the context dependency of the atomic operation execution times.

9.2 Limitations

The work presented in this thesis is limited in several ways. Predicting the ER1 system was more difficult than expected due to non-existing upper bounds for certain operations like network communication and unexpected effects of the monitoring program. Still, the ER1 system is very simple example compared to complex industrial robots or common distributed software systems. In contrast to more realistic systems, our sample scenario was a deterministic and sequential. However, the modelling language RADL and the used prediction algorithm are also suitable for non-deterministic and concurrent systems.

Another caveat concerns the predictions. Although the predictions seem to be safe, they were made under the assumption that the atomic operation execution time is independent from the execution context. At least while using the monitoring program, this assumption is violated. The violation is not severe and the predictions made are still quite accurate. For really secure predictions, such a violation is not acceptable. In practice, the system model has to be detailed enough to model the dependencies causing the violation.

The context dependencies observed during the project are twofold:

- *Component internal:* These arise by abstracting from an component state which has influence on the execution time. Basically, the component model is not detailed enough to model the component accurately. An example is the IR sensors, where multiple sensor readings within a short time increase the reading time. A sensor reading therefore seem to change the state of the sensor or at least the state of library used to read the sensors. Our model assumes the IR sensor to be stateless and fails to model the execution time depending on the inter-reading times correctly. The problem was solved in this project by avoiding to read the sensor multiple times within a short period. In general, component internal context dependencies can be solved by adding states to the component model.
- *Between several components:* These dependencies are not caused by an inappropriate component model itself, but by neglecting the influence a component has on its environment. An example is the monitoring program which shares some resources with the Robot Control Centre like the processor, memory and in particular buffers of the operating system. Thus, the monitoring program can influence the execution times of the application by changing the state of a third component, which is used both programs. The solution of such dependencies usually requires to add the missing component to the system model.

The system model used in this project is highly simplistic and fails to model many of these dependencies. It appears nevertheless to be appropriate for making good predictions of the ER1 robot system. Using a simplistic model also allowed give some insights of the effects of such simplifications.

9.3 Conclusions & Future work

The project's primary goals were achieved. A finite state machine model of a remotely controlled ER1 robot was developed and it was demonstrated that this model can successfully be used to predict the time required by the robot to solve a maze. But this project can only be considered as a fundamental, first experiment with non-functional property predictions using RADL models. The previous section points out some caveats.

Further research is necessary examining how accurate models of more realistic and complex systems can be created. This involves the evaluation of different modelling notations like UML Statecharts and their integration into a RADL model. Another approach for software systems is the generation of component state machine models from their source and perhaps some additional annotations.

Detailed finite state machines created by such approaches are expected to be huge. They are likely to cause a need for more efficient finite state machine representations, for example extended finite state machines which use bounded counters. Efficient algorithms for important finite state machine operations might also be useful. Future projects might research these issues.

Another problem of RADL is its lack of tool support. It is common knowledge that an architecture description language can only be truly useful, if it has good tool support[26]. Although some basic libraries are in development, a tool is missing which offers an integrated interface for modelling, visualisation and analysis while hiding the technical details. Studying the integration of these aspects seems to be worthwhile.

Furthermore, the common opinion that adding monitoring to an existing system is a delicate task was confirmed. Monitoring can hardly be isolated in a single component. In order to minimise the monitoring influence, it is beneficial to design the system with respect for monitoring needs. Facilities to support minimal invasive monitoring can be integrated in several levels of the system: hardware, operating system, libraries and in the application itself. If monitoring is considered during system design and the disturbance by the monitoring kept low, the monitoring code can in many cases remain in delivered system. In that way, a change of the system behaviour is excluded.

This project provides a good basis for further studies in some areas mentioned above. Affiliated projects can try to create more realistic models using appropriate notations or finite state generators. Future RADL tools can be tested with similar systems and other prediction methods can be compared to the method used in this work.

Appendix A

The FSA Model

The following sections illustrate the finite state machines model of the ER1 system. Communication between components is separated in two parts. Symbols starting with ! represent a request sent to another component. The completion of the request is represented by a symbol starting with ~. The latter symbols may end with a return value.

A.1 Application

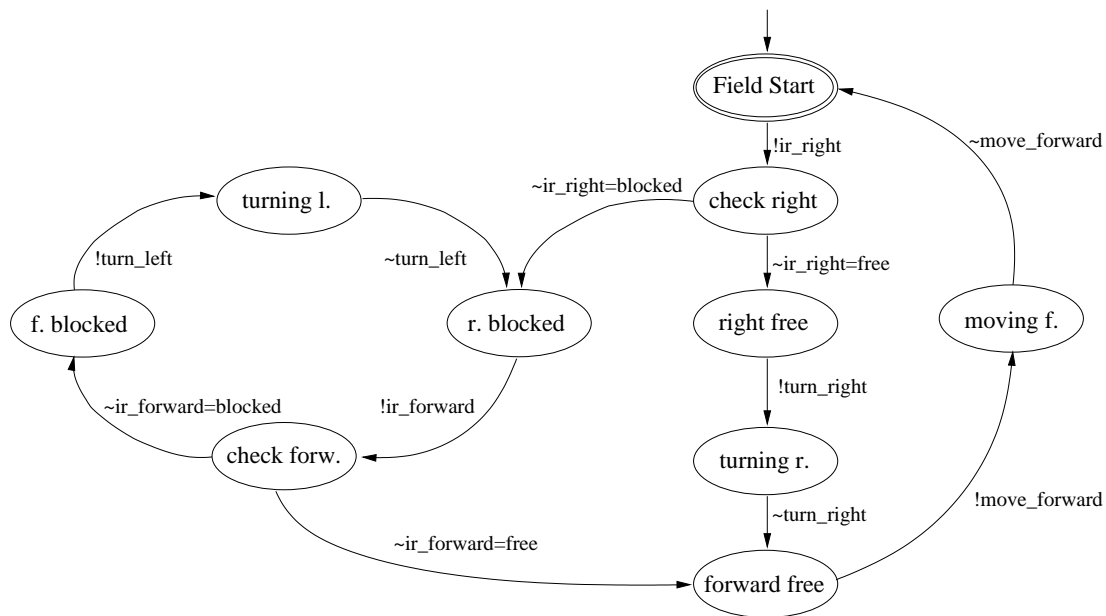


Figure A.1: The FSA model of the right wall follower algorithm implemented by the application. This FSA is a generator, that is all symbols occurring in this FSA are output symbols.

A.2 Robot

The robot component is not strictly necessary in this model. It only translates the request symbols into execution symbols. The robot component consists of two subcomponents: motors and sensors. The shuffle product of the two subcomponents creates the robot component.

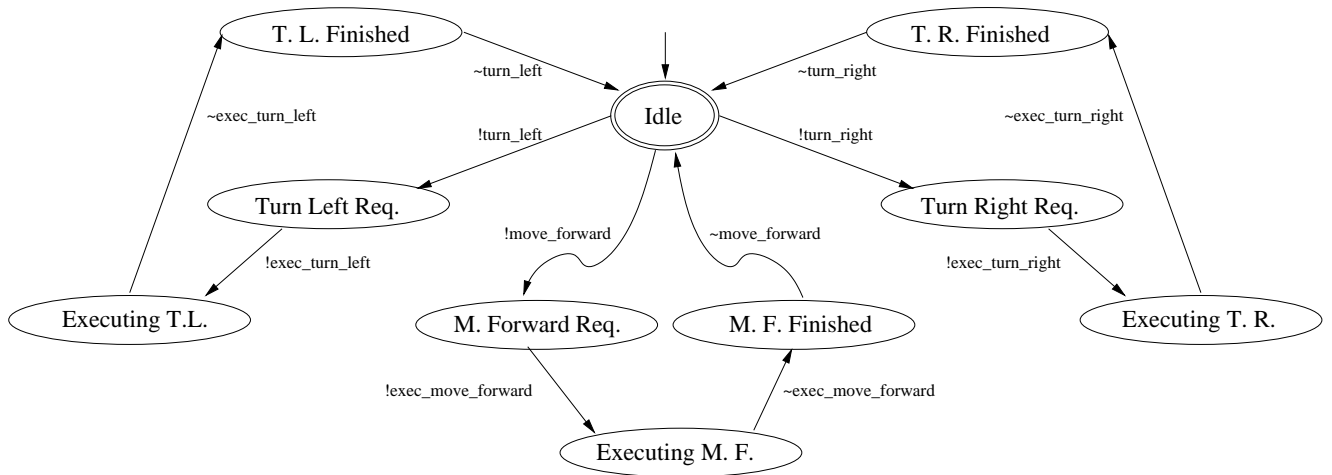


Figure A.2: The FSA model of the motor component which is a robot subcomponent.

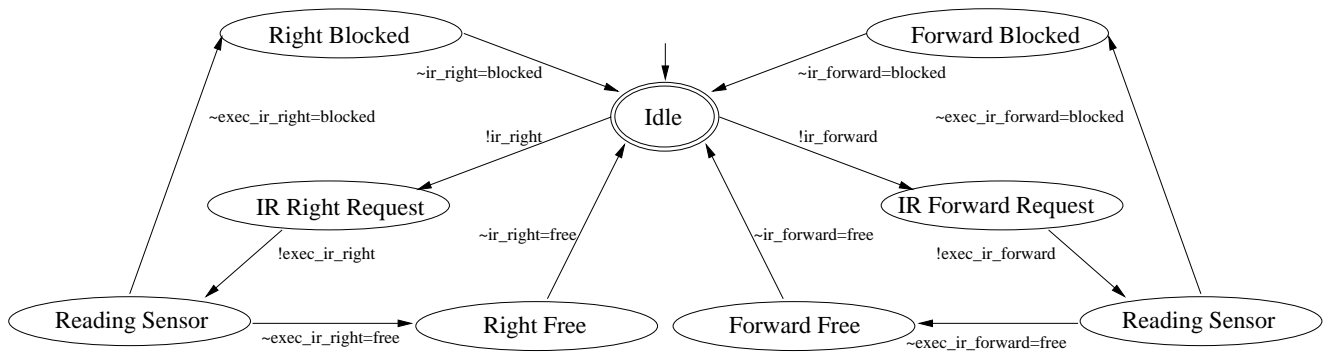


Figure A.3: The FSA model of the IR sensor subcomponent which is part of the robot component

A.3 Orientation

The FSA of the orientation component is the shuffle product of these FSAs:

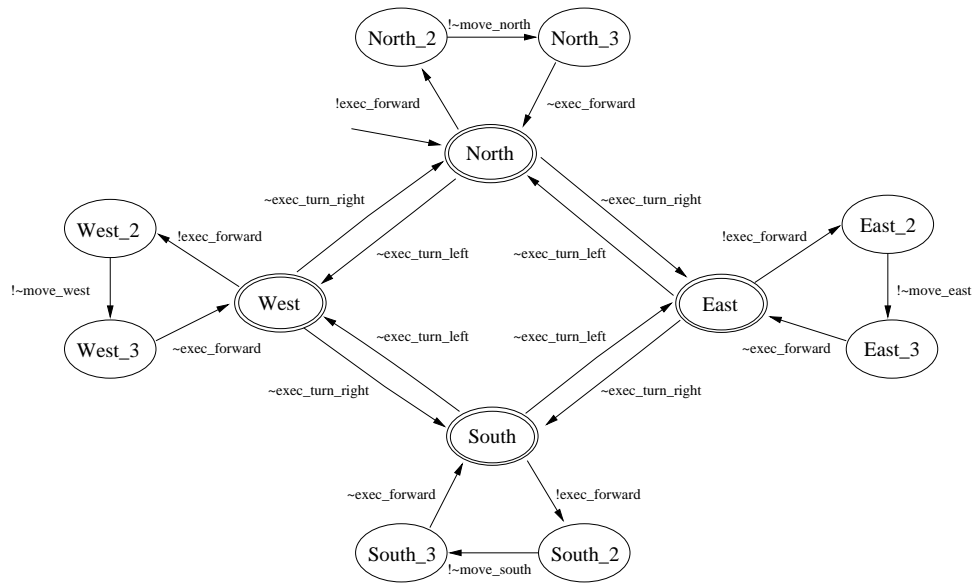


Figure A.4: A part of the orientation FSA translating the forward movement.

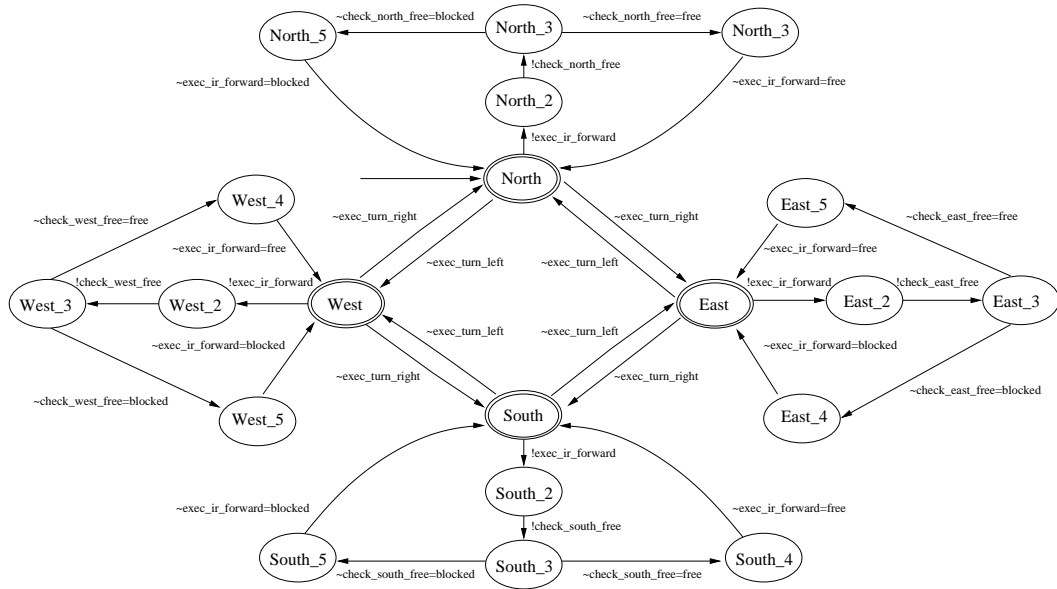


Figure A.5: A part of the orientation FSA translating the forward IR sensor readings.

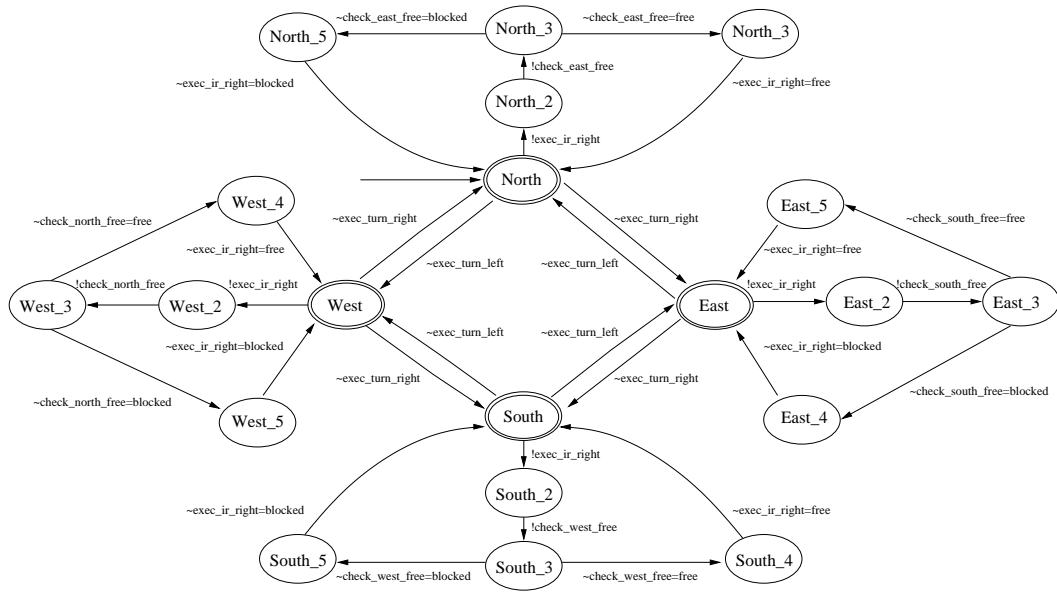


Figure A.6: A part of the orientation FSA translating the right IR sensor readings.

A.4 Map

The complete map FSA is huge. Therefore, the first figure shows a simplified version which only keeps track of the position of the robot. The actual FSA has additional transition for each state. These are shown for a specific position in figure A.8.

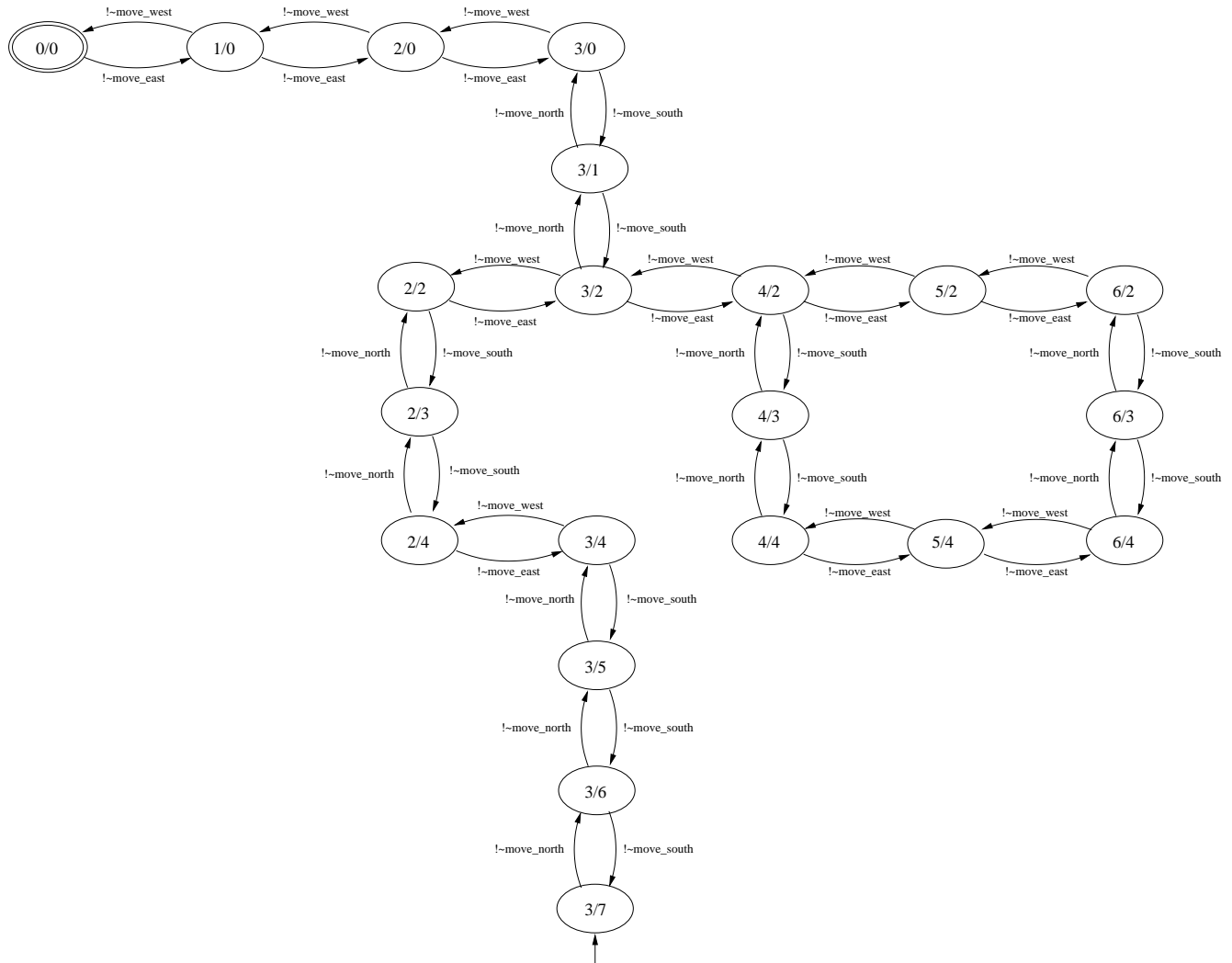


Figure A.7: The simplified FSA of the map component.

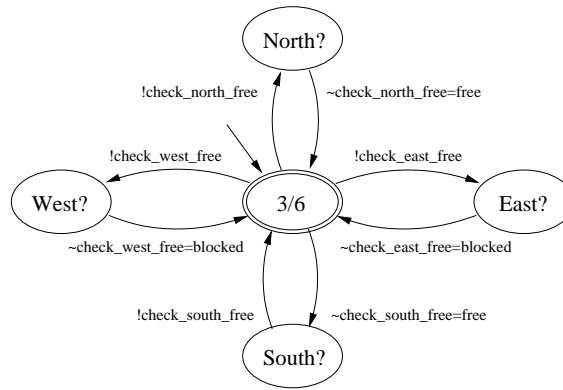


Figure A.8: Figure of a single position in the map FSA in detail.

Appendix B

Additional Diagrams

B.1 Forward movement

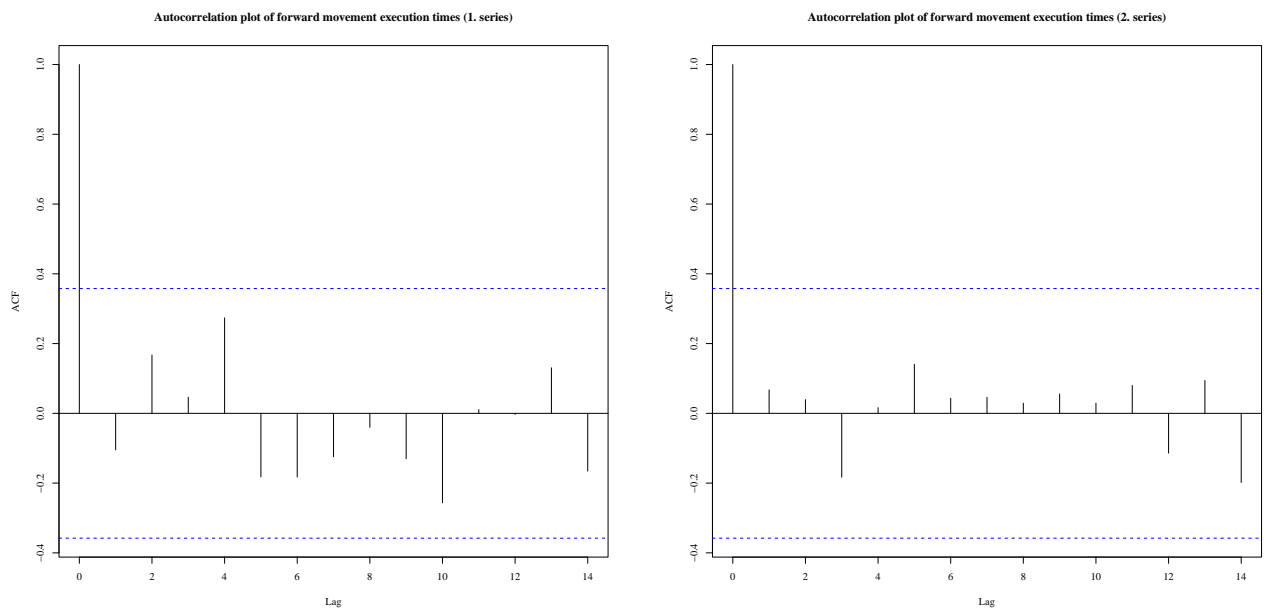


Figure B.1: Separated autocorrelation plots of the forward movement execution time measurement series

B.2 Turns

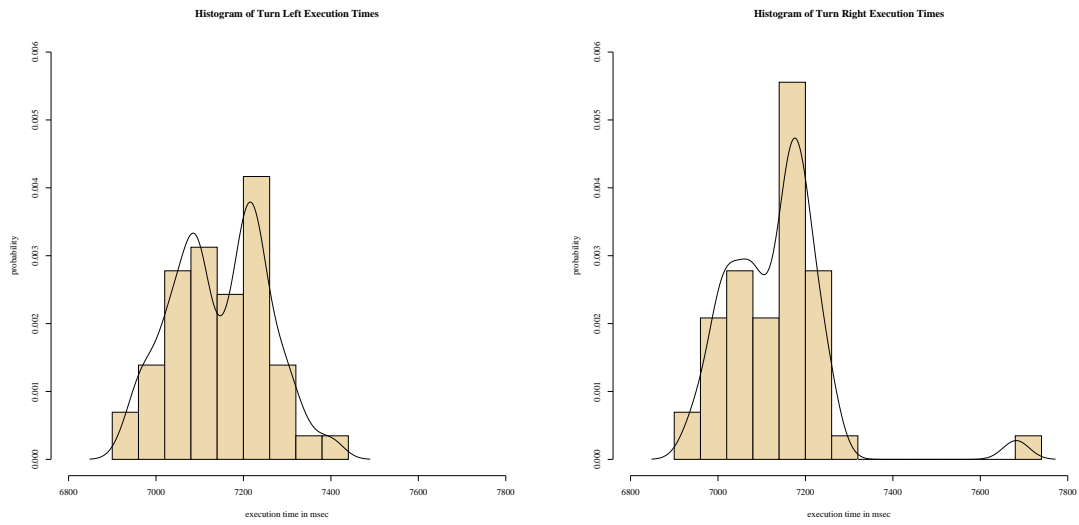


Figure B.2: Comparison of the left and right turn histograms

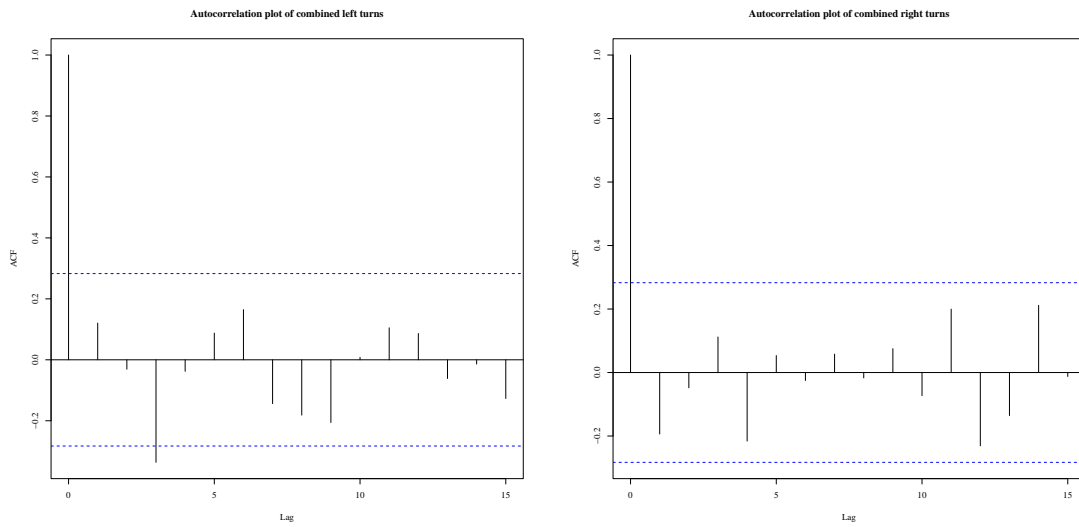


Figure B.3: A autocorrelation plots of the turn execution times

B.3 IR sensors

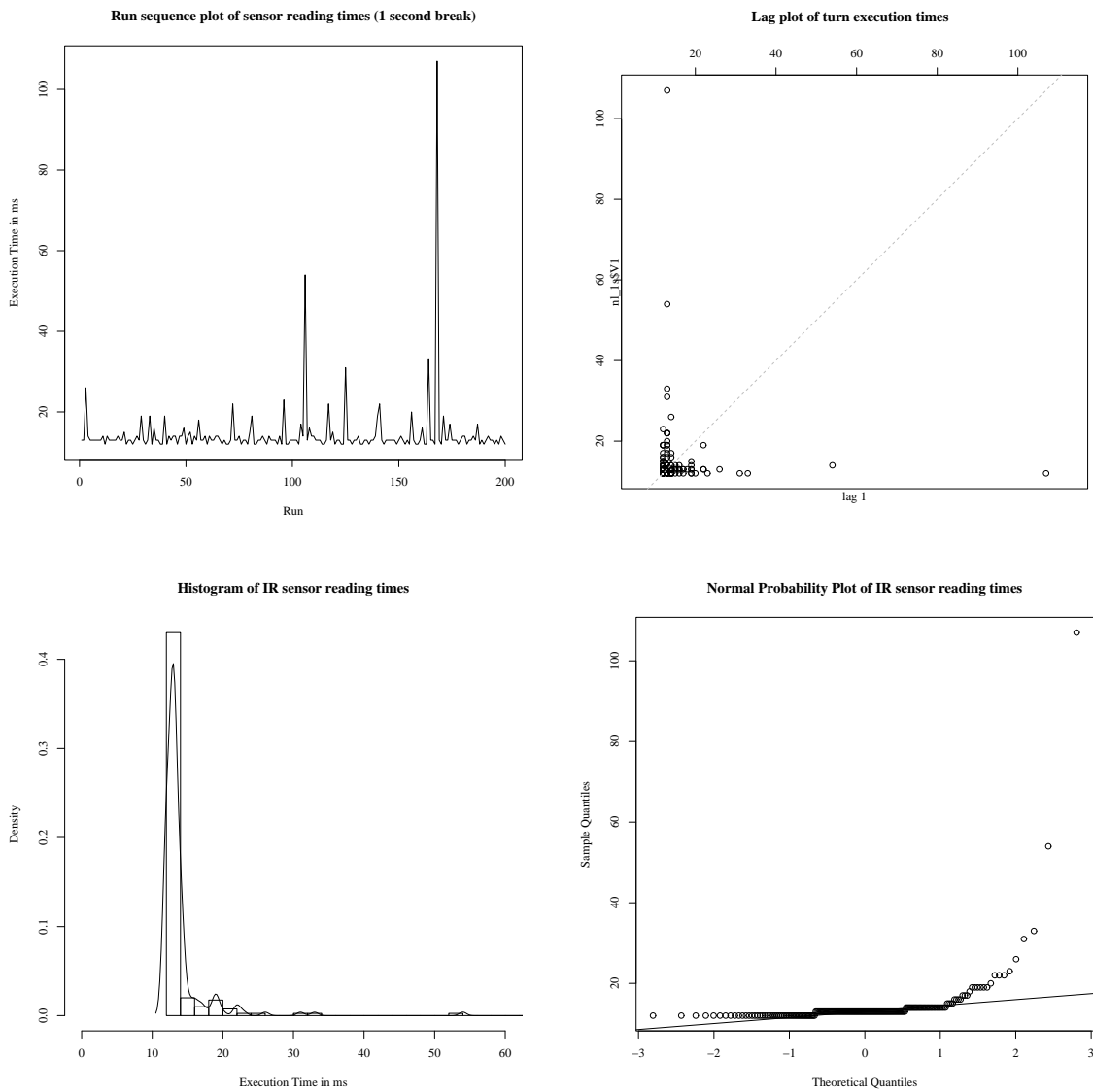


Figure B.4: 4-plot of the IR sensors reading time without monitoring and a 1 second break

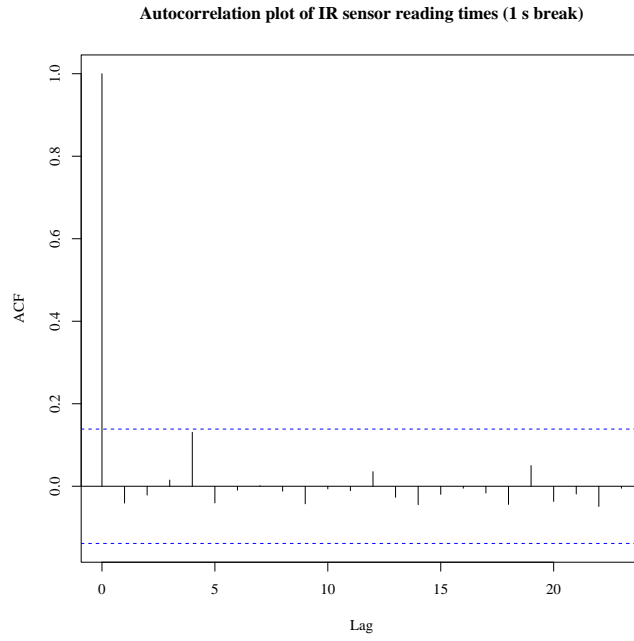


Figure B.5: A autocorrelation plot of the IR sensor reading times

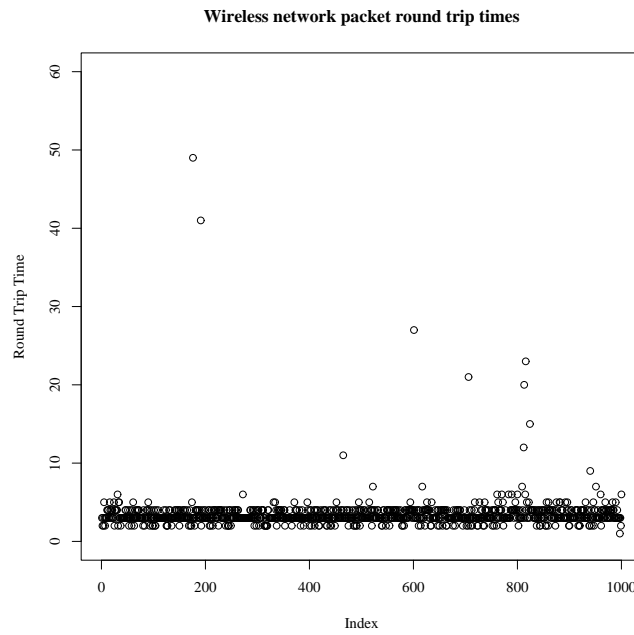


Figure B.6: Sensor reading packet round trip times in the wireless network

B.4 Monitoring influence

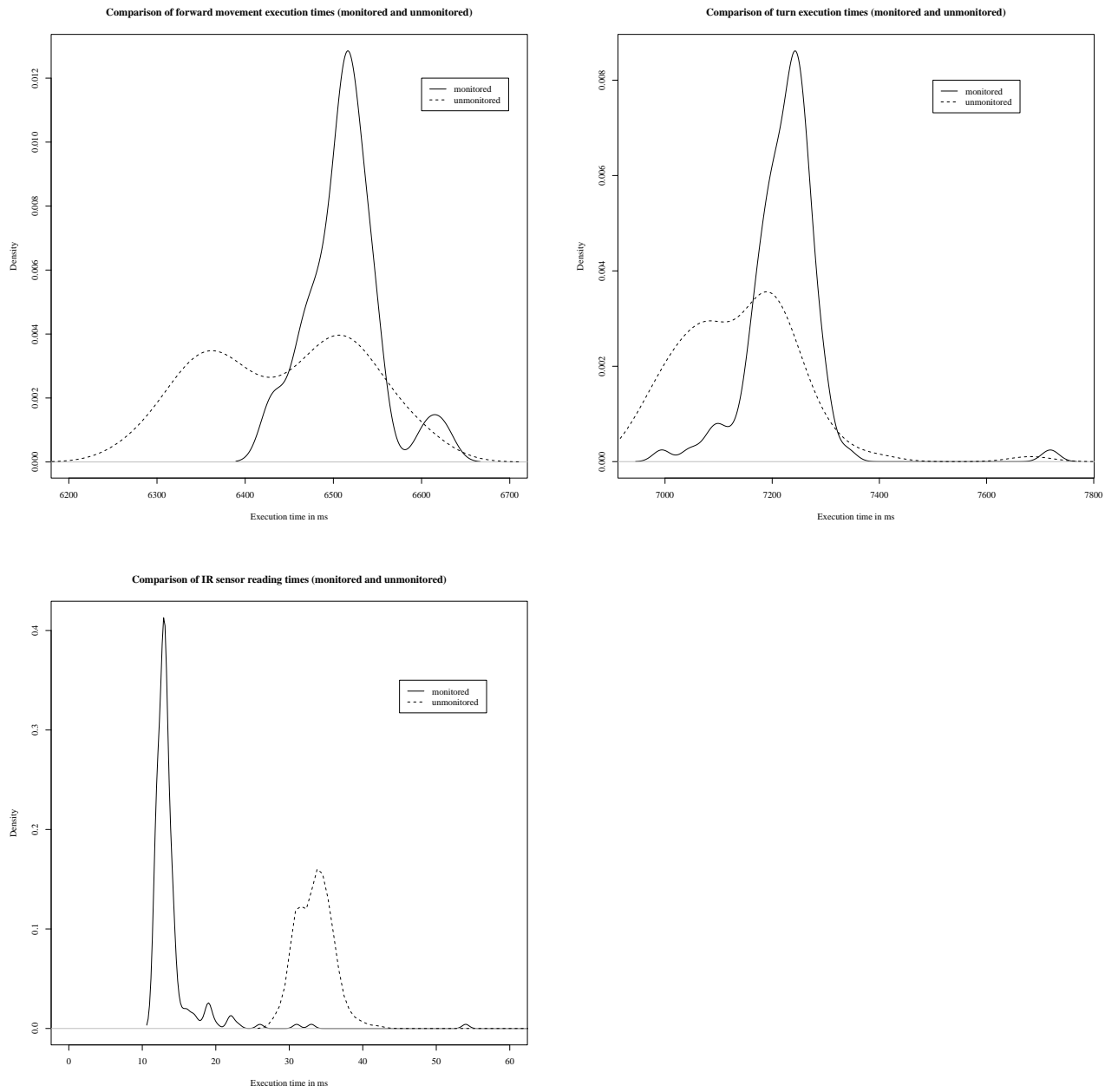


Figure B.7: Comparison of monitored and unmonitored atomic actions

Appendix C

Deliverables

All deliverables can be found on the CD-ROM attached to the thesis. The C++ programs were compiled with g++ 3.3.1 under Windows XP, but are expected to compile with any recent g++ under linux too.

- *er1/robot* :
A library providing a C++ interface to the socket interface of the Robot Control Centre.
- *er1/measure* :
The programs used to measurement the atomic action and sequence execution times.
- *er1/wall_follower* :
The program implementing the maze solving algorithm.
- *monitoring* :
The monitoring program. Synopsis: `monitor l_port dest_ip dest_port`
- *measurements* :
All log files containing the measured data, scripts to evaluate the log files and the generated plots.
- *fsa_model* :
The java implementation of the finite state machine model. It is tested with Java 1.4.2. The execution of this program requires the openECAP library of the DSSE Centre at Monash University, Australia. This library is intended to be released under a Open Source license. But until this happens, you may contact:

Prof Heinz W. Schmidt,
Distributed Systems and Software Engineering,
Monash University,
VIC 3800, Australia
email: Heinz.Schmidt@infotech.monash.edu

Appendix D

Clarification of Original Contribution

The overall idea of the project was proposed by my supervisors Prof Heinz Schmidt and Ian Peake. Beside constructive feedback and the required hardware, they in particular provided the FSA library implementing the prediction algorithm. This library is part of eCAP-CBC project[39] conducted by the Centre of Distributed Systems and Software Engineering.

The author's contribution includes

- creation of the system model (inclusive Java implementation)
- implementation of the monitoring program
- implementation of a small C++ library which provides access to ER1 robot via the socket interface of the Robot Control Centre
- design and realisation of all measurements (atomic operations, cell sequences and maze)
- choice of analysis methods and analysis of the measured data
- realisation of predictions
- comparison of predictions and measurements
- interpretation of the results

References

- [1] N. G. Leveson and C. S. Turner, “An investigation of the Therac-25 accidents,” *IEEE Computer*, vol. 26, pp. 18–41, July 1993.
- [2] J.-M. Jazequel and B. Meyer, “Design by contract: The lessons of ariane,” *Computer*, vol. 30, no. 1, pp. 129–130, 1997.
- [3] C. Heitmeyer and D. Mandrioli, “Formal methods for real-time computing: An overview,” in *Formal Methods for Real-Time Computing*, ch. 1, pp. 1–32, John Wiley & Sons, 1996.
- [4] A. Hall, “Seven myth of formal methods,” *IEEE Software*, vol. 7, pp. 11–19, Sep. 1990.
- [5] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an emerging discipline*. Prentive Hall, 1996.
- [6] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo, “Reliability prediction for component-based software architectures,” *The Journal of Systems and Software*, vol. 66, pp. 241–252, 2003.
- [7] H. W. Schmidt, I. D. Peake, J. Xie, and I. Thomas, “Modelling predictable component-based distributed control architectures,” in *Proceedings. Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, 2003*, IEEE, 2004.
- [8] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
- [9] M. K. et al., *A Practitioner’s Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Press, 1993.
- [10] P. Li and B. Ravindran, “Fast, best-effort real-time scheduling algorithms,” *IEEE Transactions on Computers*, vol. 53, pp. 1159–1175, September 2004.
- [11] P. Puschner and C. Koza, “Calculating the maximum execution time of real-time programs,” *Journal of Real-Time Systems*, vol. 1, pp. 159–176, Sep. 1989.
- [12] P. Puschner, “Guest editorial: A review of worst-case execution-time analysis,” *Real-Time Systems*, vol. 18, no. 2-3, pp. 115–128, 2000.

- [13] J. Engblom, A. Ermedahl, and F. Stappert, “A worst-case execution-time analysis tool prototype for embedded real-time systems,” in *presented at the Workshop on Real-Time Tools 2001*. available at <http://user.it.uu.se/~jakob/> (accessed on 1/7/2004).
- [14] Y.-T. S. Li and S. Malik, “Performance analysis of embedded software using implicit path enumeration,” in *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pp. 88–98, ACM Press, 1995.
- [15] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.
- [16] J. M. Spivey, *The Z Notation: A Reference Manual*. Prentice Hall, 1992.
- [17] R. Alur and T.A. Henzinger, “Logics and Models of Real-Time: A Survey,” in *Real Time: Theory in Practice*, vol. 600, pp. 74–106, Springer-Verlag, 1991.
- [18] J. C. M. Baeten, “A brief history of process algebra.” available at alexandria.tue.nl/extra1/wskrap/publichtml/200402.pdf (accessed on 3/7/2004), 2004.
- [19] I. Lee, H. Ben-Abdallah, and J.-Y. Choi, “A process algebraic method for the specification and analysis of real-time systems,” in *Formal Methods for Real-Time Computing*, ch. 7, pp. 167–194, John Wiley & Sons, 1996.
- [20] C. Petri, *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962. Also in: *Griffiss Air Force Base, Technical Report RADC-TR-65-377*, Vol. 1, pages 1-Suppl. 1. 196, english translation.
- [21] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze, “A unified high-level petri net formalism for time-critical systems,” *IEEE Transactions on Software Engineering*, vol. 17, pp. 160–172, 1991.
- [22] R. Reussner, “Counter-constrained finite state machines: A new model for component protocols with resource-dependencies,” in *SOFSEM 2002: Theory and Practice of Informatics: 29th Conference on Current Trends in Theory and Practice of Informatics*, pp. 20–40, Springer-Verlag Heidelberg, 2002.
- [23] T. A. Henzinger, J.-F. Raskin, and P.-Y. Schobbens, “The regular real-time languages,” in *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, pp. 580–591, Springer-Verlag, 1998.
- [24] R. Alur and D. L. Dill, “A theory of timed automata,” *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.
- [25] A. Mazurkiewicz, *The Book of Traces*, ch. 1. World Scientific Publishing, 1995.
- [26] N. Medvidovic and R. N. Taylor, “A classification and comparison framework for software architecture description languages,” *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

- [27] D. Garlan, R. Monroe, and D. Wile, “Acme: an architecture description interchange language,” in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, p. 7, IBM Press, 1997.
- [28] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, “Specifying distributed software architectures,” in *Proceedings of the 5th European Software Engineering Conference*, pp. 137–153, Springer-Verlag, 1995.
- [29] R. Milner, “The polyadic pi-calculus: A tutorial,” Tech. Rep. ECS-LFCS-91-178, University of Edinburgh, 1991.
- [30] J. W. Krueger, S. Vestal, and B. Lewis, “Fitting the pieces together: system/software analysis and code integration using METAH,” in *Proceedings of the 17th Digital Avionics Systems Conference*, pp. C33/1–C33/8, IEEE, 1998.
- [31] H. Schmidt, B. Krmer, and M. Papazoglou, “Compatibility of interoperable objects,” in *Information Systems Interoperability*, Research Studies Press, 1998.
- [32] S. Ling, H. Schmidt, and R. Fletcher, “Constructing interoperable components in distributed systems,” in *Proceedings Technology of Object-Oriented Languages and Systems (TOOLS) 32*, (Los Alamitos, CA, USA), pp. 274–284, IEEE Computer Society Press, 1999.
- [33] H. W. Schmidt, B. J. Krmer, I. Poernomo, and R. Reussner, “Predictable component architectures using dependent finite state machines,” in *Lecture Notes in Computer Science(Proceedings of the 9th International Workshop in Radical Innovations of Software and Systems Engineering)*, Springer-Verlag, 2004.
- [34] R. H. Reussner, *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. PhD thesis, Universität Kalsruhe, 2001.
- [35] *ER1 User Guide*. Evolution Robotics, 2003.
- [36] *ERSP 3.0 User’s Guide*. Evolution Robotics, 2004.
- [37] J. J. Tsai, Y. Bi, S. J. Yang, and R. A. Smith, *Distributed Real-Time Systems: Monitoring, Visualization, Debugging and Analysis*. John Wiley & Sons NY, 1996.
- [38] *NIST/SEMATECH e-Handbook of Statistical Methods*. electronically available at <http://www.itl.nist.gov/div898/handbook/> (accessed 21 October 2004), 2004.
- [39] D. S. S. E., Monash University, *The Extra-functional Consistency And Prediction for Component-Based Control Systems (eCAP-CBCS) Website*. <http://www.csse.monash.edu.au/~hws/ecap/> (accessed 31 October 2004).