Real-time cloud-based stochastic scheduling incorporating mobile clients and a sensor network

Cameron McOnie
Department of Industrial Engineering
University of Stellenbosch

Supervisor: Professor James Bekker

Thesis presented in partial fulfilment of the requirements for the degree of
Master of Engineering in the Faculty of Engineering at Stellenbosch University

M. Eng Industrial

March 2016

To my grandfather, J.B. McOnie.

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work; that I am the sole author thereof (save to the extent explicitly otherwise stated); that reproduction and publication thereof by Stellenbosch University will not infringe any third-party rights, and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Abstract

Scheduling within manufacturing environments is often complicated due to the complex, dynamic and stochastic characteristics such environments exhibit. These characteristics pose problems for off-line scheduling techniques as schedules, initially determined to be acceptable, may degrade or even become infeasible as the state of the system changes. On-line techniques attempt to address this challenge by performing scheduling concurrently with the manufacturing system. By reacting to system disturbances in real-time, on-line schedulers are capable of producing better schedules, or schedule control laws, when compared to off-line techniques.

This study proposes a software architecture for a simulation-based reactive scheduling system. The architecture addresses what the main components of a reactive scheduler are and how they are related. Furthermore, it describes each of the components from multiple viewpoints, i.e., logical, process, development, and deployment—predominantly using the unified modelling language. The design decisions used to arrive at architecture qualities such as scalability, modularity, and interoperability are also discussed. Particular attention is given to defining a service contract between the back-end of a reactive scheduling system and data capture and decision support devices located on the shop floor.

The proposed architecture is applied through the construction of a simulation-based reactive scheduling system, capable of reacting to real-time disturbances. The base of the system is a simulation model of a pressure gauge assembly operation. Interaction with the simulation model is done through a scheduling application server. The system also comprises of a sensor network prototype, used as means of tracking the movement of work-in-process through the assembly operation; and a mobile client, used to communicate decision support data back to the shop floor. The scheduling application server is deployed to the cloud and is exposed as a Web service for shop floor devices to consume.

An experiment that compares the effect of rescheduling using dispatching rules on the system over time is performed. It is shown that as the system state progresses, the recommended dispatching rule may change, and therefore, by embedding the associated control law into the shop floor, would result in an improvement of the manufacturing objective. This experiment illustrates the value of reactive scheduling in the presence of real-time events.

Opsomming

Skedulering in vervaardigingsomgewings is dikwels moeilik weens die komplekse, dinamiese en stogastiese eienskappe daarvan. Hierdie eienskappe veroorsaak probleme vir aflyn-skeduleringtegnieke omdat skedules wat aanvanklik aanvaarbaar was mag versleg en selfs onaanvaarbaar raak soos wat die stelsel se toestand verander. Aanlyntegnieke poog om hierdie uitdaging die hoof te bied deur skedulering samelopend met die vervaardigingstelsel te doen. Deur in reële tyd op stelselversteurings te reageer kan aanlynskeduleerders beter skedules en reëls ontwikkel in vergelyking met aflyn-tegnieke.

Hierdie studie stel die sagteware-argitektuur voor vir 'n simulasie-gebaseerde reaktiewe skeduleringstelsel. Die argitektuur identifiseer die hoofkomponente en die verwantskappe 'n reaktiewe skeduleerder. Dit beskryf elke komponent verder vanuit verskillende oogpunte, nl. die logiese-, proses-, ontwikkeling- en ontplooiingoogpunt. Hiervoor word die verenigde modelleringmetode gebruik. Die ontwerpbesluite om by argitekturele kwaliteite soos skaleerbaarheid, modulariteit en interfunksionaliteit uit te kom, word ook bespreek. Spesifieke aandag word gegee aan die definisie van 'n dienskontrak tussen die agterkant van 'n reaktiewe skeduleringstelsel en datavaslegging en besluitsteuntoerusting op die fabrieksvloer.

Die voorgestelde argitektuur word toegepas deur die opstelling van 'n simulasie-gebaseerde reaktiewe skeduleringstelsel wat in staat is om in reële tyd te reageer op versteurings. Die kern van die stelsel is 'n simulasiemodel van 'n drukmetermonteerproses. Interaksie met die simulasiemodel word bewerkstellig deur 'n skedulering-toepassingbediener. Die stelsel bevat ook 'n prototipe 'n sensornetwerk wat gebruik word vir die naspoor van werk-in-proses deur die monteeraanleg en 'n mobiele kliënt wat gebruik word om besluitsteundata aan die fabrieksvloer te kommunikeer. Die skedulering-toepassingbediener is in die elektroniese wolk ontplooi en word aan fabrieksvloertoestelle as n webdiens ontbloot.

'n Eksperiment wat die effek van herskedulering met sekere reëls op die stelsel oor tyd ondersoek, is uitgevoer. Dit word getoon dat soos wat die

stelseltoestand ontvou, die aanbevole skeduleringreël kan verander, en deur sulke sulke reëls op die fabrieksvloer in te sluit kan die vervaardigingsdoelwit verbeter word. Hierdie eksperiment illustreer die waarde van reaktiewe skedulering in die teenwoordigheid van reële-tyd gebeure.

Acknowledgements

I would like to express my gratitude to all those who encouraged me throughout this thesis. In particular, I would like to provide a special thank you to the following people and organisations:

- Professor James Bekker, my supervisor, for his guidance, financial support, and incredible patience.
- My grandfather, for his financial support and unwavering confidence in my ability.
- My parents, for affording me the opportunity to study.
- My sister, for her constant assurances.
- My office colleagues, for their witty remarks, and candid advice.
- The industry partner, for allowing me to conduct a simulation study at their premises.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations and Acronyms

AD	Architecture description
ADL	Architecture description language
API	Application programming interface
DFD	Data flow diagram
DHCP	Dynamic host configuration protocol
DTO	Data transfer object
EDD	Earliest due date
ERD	Earliest release date
FIFO	First in first out
GUI	Graphic user interface
IDE	Integrated development environment
IO	Input/Output
IQR	Inter quartile range
LNG	Least number of gauges
LPT	Longest processing time first
MPC	Model predictive control
MVC	Model view controller

MVE	Minimal viable example
OOAD	Object-orientated systems analsis and design
QA	Quality Assurance
RSA	Reactive scheduling application
RSS	Reactive scheduling system
SDK	Software development kit
SDLC	Systems developoment life cycle
SDM	Software development methodology
SIRO	Select in random order
SIT	System integration testing
SME	Subject matter expert
SOA	Service orientated architecture
SPT	Shortest processing time first
UML	Unified modelling language
WIP	Work-in-process
WSN	Wireless sensor network

Greek Symbols

ϕ_i	Eligible tasks at workstation i
δ	Indifference zone
$\hat{\Theta}_k$	Point estimate for scenario k
α	Probability of correct selection

Roman Symbols

$\hat{L}$	Cummulative job lateness
D_k	Decision variable for scenario k

$\hat{E}_j$	Estimated earliness for job j
$\hat{F}_j$	Estimated flow time for job j
$\hat{L}_j$	Estimated lateness for job j
$\hat{C}_j$	Estimated completion time for job j
h	Half-width
$\bar{\hat{F}}$	Mean job flow time
W	Mean work-in-process
M	Median of the performance measure
n	Number of replications
$\hat{U}$	Number of tardy jobs
Y_{kn}	Observation n for scenario k
O_{ij}	Operation to be performed at processor i for task j
X_{km}	Random variable m for scenario k
RP_i	Rescheduling profile
$R_{\max}$	Response time constraint
k	Number of simulation scenarios
$\hat{C}_M$	Schedule makespan
T_O	Simulation model start time
T_E	Simulation model end time
S_t	State of the factory floor at time t
T_i	Task with index i
TP	Throughput rate
p_i	Workstation with index i

Chapter 1

Introduction

1.1 Background

Manufacturing has traditionally been, and remains to be, a complex engineering endeavour. The challenge of organising and coordinating people, material, equipment, and information toward a manufacturing goal demands considerable time and effort. Though many of these challenges can be addressed by careful planning during the design phase of the manufacturing system, certain challenges continue to persist over the operational phase. For many manufacturing systems, scheduling is just such a challenge; and can be attributed to the complex, dynamic, and stochastic environments exhibited by such systems.

Traditional approaches attempt to address the scheduling problem by creating and evaluating schedules prior to production commencing. Inevitably however, unexpected events, e.g. expedited orders, arise during the production process, disturbing or potentially invalidating the existing schedule. In such scenarios managers must react quickly by selecting a new or revised schedule to ensure production continues—while still maintaining a good level of performance. Real-time scheduling systems have attempted to address the shortcomings of this traditional approach by performing scheduling concurrently with the production process. In doing so, the future impact short-term decisions, e.g. dispatching rules, can be estimated and used to intelligently control the process.

Due to the complexity and temporal constraints of real-time scheduling, software-intensive systems must be constructed to execute and support these systems. Furthermore, the realisation of such systems involves the stitching together of numerous

distributed hardware and software components, e.g. data collection devices, experimentation engines, control mechanisms, etc. Naturally, the architectures of such systems, and the qualities of those architectures, e.g. usability, scalability, maintainability, etc., play an important role in their eventual success. Many recent technological advances, including: cloud computing resources, the ubiquity of mobile devices, and the improved capabilities of sensor networks, have all offered potentiality new ways of designing real-time scheduling systems, and correspondingly, afforded the opportunity for new software architectures to be created to support them.

1.2 Aim and objectives

The aim of this thesis is to design and develop a prototype of a real-time simulation-based scheduling system. Supporting this aim, the *research intent* of this thesis is to propose a software architecture for the system to be developed.

To achieve the stated aim and intent, the following objectives were set:

1. Acquire domain knowledge on simulation-based real-time scheduling systems.
2. Specify a set of generic requirements for a simulation-based real-time scheduling system.
3. Construct a simulation model to serve as the foundation for the study.
4. Select a software development life-cycle to govern the software creation process.
5. Design and develop a real-time scheduling system incorporating mobile clients and a sensor network.
6. Design and document the architecture for the system to be developed.

1.3 Research methodology

The research effort in this thesis begins with an investigation into the landscape of real-time scheduling systems, specifically within manufacturing environments. Attention will be given to the common characteristics and architectures of these systems. The outcome of this investigation will then used to formulate a set of requirements for the identification of a suitable industry partner. The purpose of the industry partner is to serve as a grounding point for the study, and for the construction of a simulation

model, which will form the basis for the development of a real-time scheduling prototype. This initial effort will follow an exploratory methodology. Specific questions about the characteristics and functioning of reactive scheduling system will be asked. Answers to these questions will become more precise and refined as the prototype of the system is built and practical knowledge is gained.

Laplante & Ovaska (2012) advise that designers who intend to create a real-time solution become familiar with a range of disciplines, including: computer architecture, organisation architecture, operating systems, programming languages, systems and software engineering, as well as performance evaluation and optimisation techniques. Thus, research into potential technologies, languages, paradigms, and methodologies, needed to build each identified component, will be performed within the context of rescheduling.

Once an investigation into the characteristics and functioning of real-time scheduling architectures is complete, a set of generic business requirements will be specified. These requirements will define the project's scope at a high level. At this point the industry partner's operation should be understood in sufficient detail for these requirements to be narrowed to define the scope of the real-time scheduling prototype.

The construction of a software intensive system, such as a real-time scheduling system, demands that a software development life-cycle be used to guide the design and development effort. This ensures that the software has a well defined feature set and a clear goal regarding what it should achieve. For this reason, an investigation into the merits and applicability of popular software development life-cycles, with respect to the construction of a real-time scheduling system, will be performed. As an outcome of this process a development life-cycle will be chosen to guide the construction of the prototype.

Since the purpose of the real-time scheduling prototype is to provide tangible input into the specification of a software architecture, it is necessary for an understanding to be gained on the methods and tools of software architecture documentation. Thus, both a process for arriving at a 'good' architecture, and a formalised means of documenting this architecture will be examined. The selection of a software architecture description language will allow the design decisions made during the construction of the prototype to be effectively communicated.

The final portion of this research involves experimentation with the constructed simulation model. A quantitative analytical approach is applied with the intent of

obtaining information about the simulation model's response. This systematic approach will allow valid conclusions to be drawn about the experiment's output.

1.4 Structure of the document

A brief summary of each of the chapters in this thesis is given below.

Chapter 1 ***Introduction***

The background to this study was provided in this chapter. The direction of the thesis was set via the aforementioned aim and intent, lastly, the path to achieving this was detailed via the thesis objectives.

Chapter 2 ***Real-time stochastic scheduling***

An overview of real-time stochastic scheduling is given with the purpose of contextualising this study. Literature on real-time scheduling systems is presented, the basic components are described, and vocabulary relating to real-time scheduling is defined. The chapter ends with the specification of a set of general requirements for a reactive scheduling system.

Chapter 3 ***Simulation model of a pressure gauge assembly operation***

Simulation as a scheduling technique is introduced. Dispatching rules are classified and their applicability is demonstrated using a tangible example. An industry partner is identified and discussed, after which the development of a simulation model, the body of the chapter, is stepped through in detail.

Chapter 4 ***Selection of a software life-cycle model***

Three prominent software life-cycle models are described and contrasted. The unified process, a derivative of the iterative and incremental life-cycle, is proposed as the methodology of choice for this thesis; allowing for the planning, structuring, and control of the software creation process. The UML is introduced, and its popularity as tool for documenting software architectures is highlighted.

Chapter 5 ***Tracking WIP using a sensor network***

In this chapter, a sensor network prototype capable of capturing internal manufacturing disturbances is described. Its reliance on RFID as a work-in-process tracking mechanism is discussed, and the communication between the sensor network and the reactive scheduling application is defined. Additionally, the application of the UML modelling process to the design of prototype is demonstrated.

Chapter 6 ***System management and control using a mobile client***

The use of a mobile client as a means of both capturing external manufacturing disturbances and presenting decision support data is described. Application of the UML modelling process as input to the mobile client's software construction phase is discussed; and lastly, the integration of the mobile client with the scheduling application's application programming interface is introduced.

Chapter 7 ***Development of the reactive scheduling application***

The structure and functioning of the reactive scheduling system are presented in this chapter. Specific attention is given on the simulation optimisation engine, and its reliance on the Kim Nelson procedure (Kim & Nelson, 2001) for scenario selection. The RESTful architectural style is described and its use in exposing a Web service to the sensor network and mobile client is discussed. Finally, the use of cloud computing as an enabler of the architecture is described.

Chapter 8 ***Experimentation and results***

In this chapter, the research intent of this thesis is addressed by presenting a summary of the software architecture detailed throughout the preceding chapters. The results from experimentation on the reactive scheduling system using dispatching rules are discussed.

Chapter 9 ***Conclusion***

In this final chapter, a summary of the document is provided and the thesis outputs are mapped to the aim and objectives of this study. Finally, areas for potential future research are identified.

Chapter 2

Real-time stochastic scheduling

This chapter begins by introducing real-time scheduling and defining some of the key concepts of the field; following this, some important characteristics of real-time scheduling systems are defined. The use of simulation within real-time scheduling system is presented and along with a discussion on a few of the basic principles. The chapter concludes with the specification of a set of generic requirements for a real-time scheduling system. These concepts, principles, and characteristics of this chapter lay the foundation for the system developed in this thesis.

2.1 Introduction to real-time stochastic scheduling

Scheduling attempts to improve manufacturing process by assigning jobs to resources to timeslots—structuring the daily operation in some intelligent manner. In traditional scheduling, this assignment is performed off-line (or prior to production commencing). however recent advances in computing power, network bandwidth, and scheduling theory enabled on-line (real-time) scheduling to emerge as a practicable field. Such real-time scheduling systems have lead to the more efficient control of complex and dynamic manufacturing environments as on-line control laws eliminate many of the simplistic assumptions of off-line scheduling Jaoua *et al.* (2012). Real-time scheduling circumvents many of these assumption by operating on current data, reducing uncertainty and thus increasing the accuracy of the model's predictions. The term 'real-time' is used throughout this thesis and is therefore worth formally defining.

2.1.1 Definition and classification of a real-time system

Real-time systems span several domains and are found in almost all industries, with typical examples being: space systems, networked multimedia systems, and embedded automotive electronics. Broadly speaking, they exist for the purpose of transforming inputs into outputs in some time-bound manner. Real-time systems are synonymous with real-time control systems in which inputs are collected by sensors and outputs are used to drive actuators. This process is illustrated in Figure 2.1.

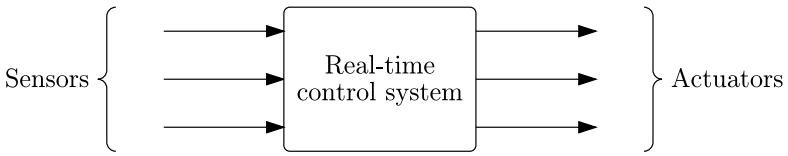


Figure 2.1: Real-time control system.

Formally, a **real-time system** is one whose logical correctness is based on both the correctness of the outputs and their timeliness (Laplante & Ovaska, 2012). Within a real-time system's realisation exists an inherent delay between the presentation of inputs (requests) and the appearance of outputs (responses). The duration of this delay is termed the system's *response time* (R_t), and is an important metric of such systems. The response time generally follows some random variable arising from stochastic elements in the system's control loop—such as network latency for example.

Classification of real-time systems is based on the manner in which the system reacts to responses that fall outside of a predefined specification, know as a response-time constraint ($R_{\max}$). In other words, real-time systems are characterised according to the effect on the system if a deadline is missed. Kopetz (2011) and Laplante & Ovaska (2012) provide the following three classifications:

1. Hard: Failure to meet a single response-time constraint will result in complete or catastrophic failure, e.g., vehicle cruise control. The distribution of R_t must fall with in the response time constraint $R_{\max}$ (see Figure 2.2).

2. Firm: Failure to meet an arbitrary, yet finite, amount of deadlines is permissible, and will not result in catastrophic failure, e.g., lawn-mowing robot. The probability of R_t exceeding $R_{\max}$ must fall within some acceptable value (see Figure 2.3).

3. Soft: Performance is degraded if deadlines are missed, but catastrophic failure does not occur, e.g., internet voice calling, The distribution of R_t may fall outside the response time constraint $R_{\max}$ (see Figure 2.3).

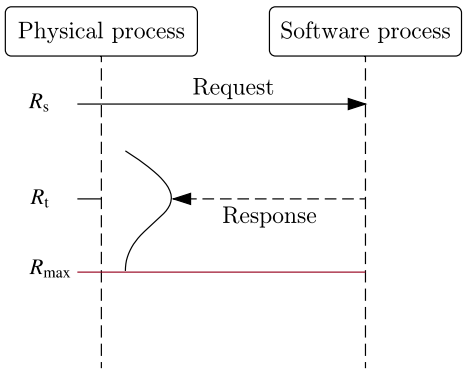


Figure 2.2: Hard real-time.

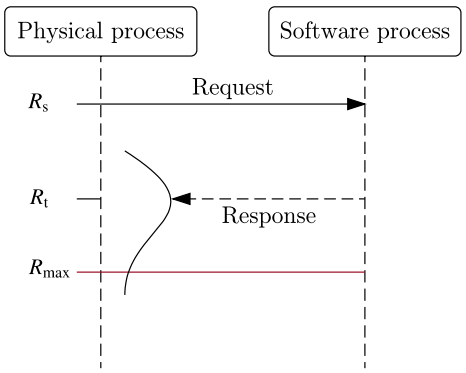


Figure 2.3: Soft and firm real-time.

2.1.2 Real-time scheduling in manufacturing environments

Real-world manufacturing systems are inherently dynamic and inevitably have unpredictable events that alter their state. For example, machines may fail as a result of wear and tear, job priorities may change due to expedited orders, or new jobs may be added to the system. Scheduling in a such an environment is difficult since the system is continually undergoing state transitions. Each of these transitions may render a previously feasible schedule infeasible, or, simply offer up the possibility that a better schedule now exists.

In recent times, renewed emphasis has been placed on real-time techniques in manufacturing, due to the tangible benefits that have seen in industry (Jaoua *et al.*, 2012). By scheduling in real-time, and taking into account disturbance events on the factory floor as they arise (termed *reactive scheduling*), the scheduler has a better chance of producing a good schedule with respect to some objective, when compared to an off-line scheduler that is unable to react to the system. This opportunity of a potentially better schedule arising at each discrete state change of the system is the driver behind the field of reactive scheduling in manufacturing.

The term real-time scheduling and reactive scheduling are synonymous, in that they both refer to scheduling in the presence of real-time event, i.e. concurrently with the operation of the system. From this point on ward the term reactive scheduling will be used.

2.2 Characteristics of a reactive scheduling system

The characteristics of a reactive scheduling system (RSS) vary along many dimensions. Typically, an RSS is designed to suit a particular manufacturing environment and is therefore tailored to meet the specific scheduling requirements of that manufacturer. This fact makes it hard to present a generic RSS which captures the all implementation instances seen in industry. Nevertheless, certain common characteristics can be still be identified. In keeping with the proposed research methodology (see Section 1.3), a set of questions were formulated based on an initial overview of rescheduling literature. These questions aim to elucidate some of the common characteristics of a RSS.

1. What is the difference between a static and dynamic rescheduling environment?
2. What rescheduling strategies are common in industry?
3. What role does the rescheduling policy play?
4. What constitutes a rescheduling technique?
5. What defines a rescheduling method?

These questions are addressed in the sections below, and are supplemented throughout the document. References to appropriate authors is also given.

Rescheduling environment

The set of jobs that is considered by the rescheduling system at the point when rescheduling occurs is defined as the scheduling environment Vieira *et al.* (2003), these are:

1. Static: A finite set of jobs is considered.
2. Dynamic: A infinite set of jobs arrive over a rolling time horizon.

The rescheduling environment is independent of the real system's environment, and refers only to the manner in which the model of the system is constructed. For example, many scheduling models consider a static environment, where as the underlying real system is dynamic.

Rescheduling strategy

Characteristics describing the way the rescheduling process is handled is captured by the rescheduling strategy. The two primary strategies are:

1. Completely reactive scheduling (Suwa & Sandoh, 2012): No schedule is generated in advance. Decisions, based on some control law, are made locally and in real-time and in response to some unexpected event.

2. Predictive-reactive scheduling (Ouelhadj & Petrovic, 2009): An initial baseline schedule is generated, system disruptions trigger rescheduling in which corrections to the schedule are made, e.g. right shift rule.

Both strategies fall under the category of *on-line* scheduling, where decisions about which tasks should be executed next are made at runtime, i.e. as the production process executes[1] (Błażewicz *et al.*, 2007).

Rescheduling policy

Both aforementioned strategies are triggered in response to real-time events. The source of those events and the real-time scheduler's reaction to those events is governed by the rescheduling policy. In other words, the policy determines which events are eligible for transformation into a rescheduling point, i.e. time instances at which the scheduler will be invoked. RSS's are invoked by events categorised as originating from two sources: the arrival of a point in *time*, and the occurrence of an *event*, resulting in three policy classifications Pop *et al.* (2004):

1. Time-driven.

2. Event-driven.

3. Hybrid.

In time-driven scheduling systems the rescheduling points are independent of the system's operation, and are therefore deterministic. Whereas in event-driven systems the rescheduling points exhibit stochastic behaviour—due to the stochastic nature of the event sources. Manufacturing environments are prime examples of systems with stochastic events, e.g., job arrivals, machine failures, and equipment shortages, etc. Hybrid policies respond to both time and event triggers.

[1] This is in contrast to *off-line* scheduling where decisions about resource allocation are made at compile time, i.e. before production begins.

Rescheduling technique

This classification applies to the technique used to perform the scheduling, i.e. the technique responsible for analysing the scheduling problem and generating some actionable instruction which is issued to the factory floor. Examples include: rule-based expert systems (Chongwatpol & Sharda, 2013), genetic algorithms (ManChon *et al.*, 2011), numeric computer simulations (Jaoua *et al.*, 2012; Zhang *et al.*, 2009), and enumerative approaches such branch and bound, and dynamic programming, to name a few.

Rescheduling method

Framinan *et al.* (2014) defines a scheduling *method* as a formal procedure that can be applied to any instance of a scheduling model with the goal of producing a feasible schedule, and which attempts to improve some performance measure of the schedule. Examples include the right-shift operation where schedule elements are shifted right in order to maintain precedence constraints, and dispatching rules where tasks are ranked at workstations according to a predefined rule.

Having answered these questions a better understanding of some of the fundamental concepts of the rescheduling domain was gained.

2.3 Overview of a simulation-based reactive scheduling system

The use of simulation as a rescheduling technique has become a popular field of research with a wide range of applications. For example, Jaoua *et al.* (2012) investigated a simulation-based RSS for the dispatching of mining trucks using a completely reactive strategy and a time-driven policy (120 s). Zhang *et al.* (2009) applied an RSS to a semiconductor wafer fabrication facility using a completely reactive strategy with an event-driven policy. Krige (2008) developed an event-driven RSS for a make-to-order job shop in which the schedule was revisited upon the arrival of new orders at the system boundary. In each study respectable schedule improvements were shown.

This section provides a definition, discusses the basic principles behind, and presents a feedback control interpretation of simulation-based RSS. Specifically, the RSS is discussed with a manufacturing setting as it is the area of research for this thesis.

2.3.1 Definition of a simulation-based RSS

By extension of the definition of a real-time system by Laplante & Ovaska (2012) in Section 2.1.1, a simulation-based reactive scheduling system is defined as follows:

> A **simulation-based reactive scheduling system** is one in which a simulation model of the real system forms part of the control loop and whose scheduling capability is based on outputs from the simulation model and their timeliness.

2.3.2 Basic principles

The principle behind a simulation-based RSS is in estimating and comparing the future effect that a set of decision variables will have on a schedule-related performance measure of the system. By generating a set of alternative future scenarios, as shown in Figure 2.4, the scenario that performs the best with respect to some performance measure can be be identified, and its corresponding decision variables transformed into some form of control instruction for execution on the shop floor. By repeating this procedure at discrete points in time the shop is continually assured that it is employing a reasonably good control law.

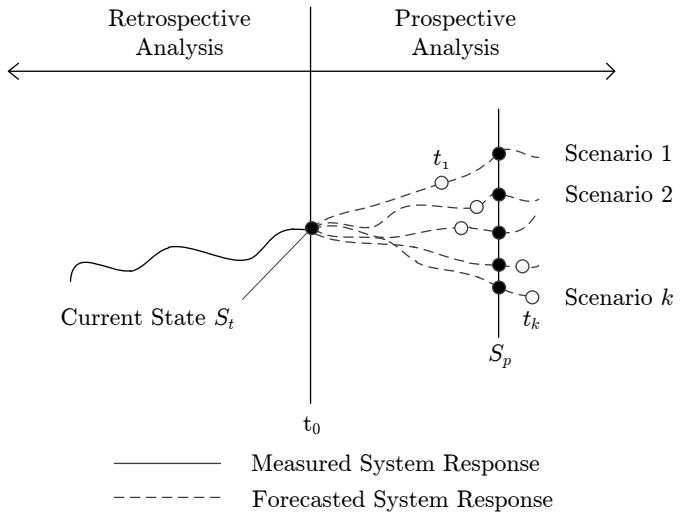


Figure 2.4: Real-time scheduling scenario generation (Banks, 1998).

More specifically, the procedure is called when the real-clock arrives upon a rescheduling point t_0 (Figure 2.4). At this point the current state S_t of the manufacturing system is fed into a simulation model of the system. This model is used to simulate k potential

short-term state trajectories (scenarios) of the system. All trajectories estimate one or more performance measures of the system (e.g. throughput rate) at some future point S_p, or the occurrence of an event t_k (e.g. a job's completion time). The best trajectory can then be identified and the decision variable(s) of that scenario transformed into a schedule control law which is embedded into the shop floor for execution. The reactive nature of real-time scheduling means that different scenarios are constantly evaluated as the state of the real-system evolves and as new rescheduling points are determined. Thus at any rescheduling point a different control law may potentially be prescribed to the shop floor by the simulation application.

Banks (1998) identified numerous challenges facing the implementation of real-time simulation systems. These were related, primarily, to the limited feature set of available simulation software and the insufficient computing power required to perform on-line simulation. For example, a key concern was in ensuring that the essential number of simulation replications, across all scenarios, could be completed faster than a real-time clock, i.e. meeting the response time constraint $R_{\max}$ (Section 2.1.1). However, simulation software and computing resources have advanced sufficiently to permit the implementation of reactive simulation systems.

2.3.3 A feedback control perspective of a reactive scheduling system

To further understand the components and functioning of a reactive scheduling system, it is useful to explore the system from a feedback control perspective. Simulation-based RSSs are synonymous with real-time control systems as they consist of a process whose output is to be controlled, and operate concurrently with a real world process. An RSS possess traits similar to model predictive control (MPC), since a model of the process is used to predict the future evolution of the process via experimentation with different control instructions, which are ultimately used to control the real process.

One differentiating property of an RSS is that no external reference signal is provided to the control system, this is in contrast to say, an air-conditioning system where a reference temperature (set point) is provided for the control system to continuously track. An RSS internally generates a set reference signals (i.e. estimated performance measures) each corresponding to a simulated scenario. The internal reference signal exhibiting the best relative fitness is selected (see Section 2.3.2) and is then used to govern the process. In doing so, the system is continually attempting to improve the scheduling objective by selecting the signal (simulation scenario) with the best output.

The logical relationships and the flow of information through the control system components are shown in Figure 2.5. A short description of each component from an MPC perspective is provided thereafter.

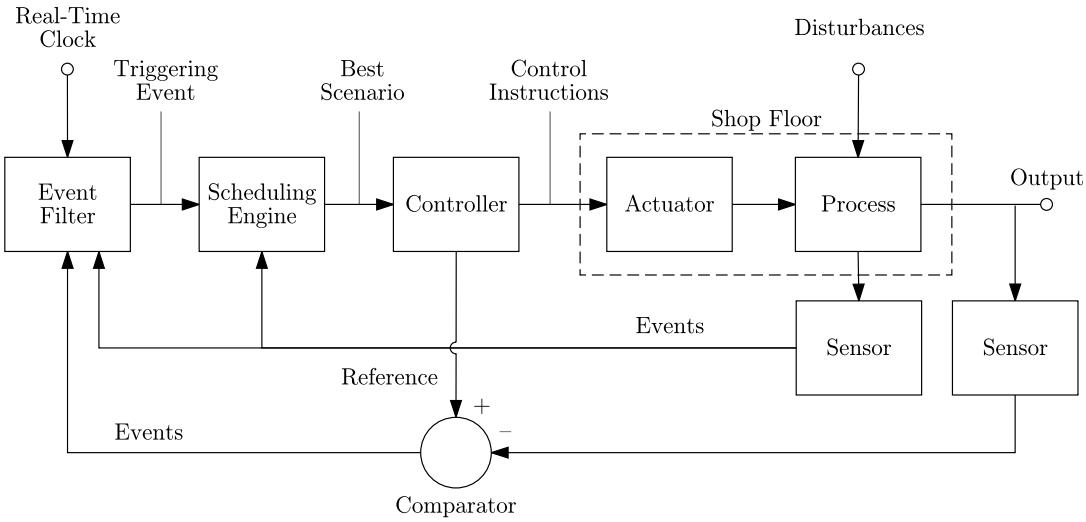


Figure 2.5: Feedback control perspective of a simulation-based RSS.

- **Event filter**: Responsible for handling and interpreting the disturbances (events) generated by the system, e.g. WIP movement. The event filter adheres to the rescheduling *policy*, only permitting certain 'white-listed' events to invoke the scheduling engine.
- **Scheduling optimisation engine**: By running a series of independent simulation scenarios, all initialised with the current state of the system, the scheduling engine estimates potential state trajectories of the system. The scenario displaying the highest contribution to the scheduling objective at the end ot of the scheduling horizon is then passed on to the controller for analysis. The type of simulation analysis that is performed is dependent on the rescheduling *environment*.
- **Controller**: The controller is responsible for transforming scenario output data received from the scheduling engine into a usable control instruction. This instruction must be in a format understandable by the actuator which might be a piece of automation equipment or a human operator. Control instructions vary according to the rescheduling *strategy* in place.

The event filter, scheduling engine, and controller are software components and thus have no physical presence on the shop floor. Rather, their logic is encoded within a software application.

- **Actuator**: Actuators convert control instructions into a physical action. For example, a schedule control law received from the controller may translate into instructions informing workstation operators as to which jobs they should attend to next. The actuators form the forward link between the software process and the physical process (i.e. the shop floor).

- **Process**: The manufacturing facility (comprised of the workstations, equipment, operators, and the set of jobs to be operated on) is referred to as the process. Disturbances affect the process resulting in discrete state changes. The purpose of an RSS is to manipulate the process using control instructions, thereby aligning the output of the process with the best simulated state trajectory (or reference signal). In other words the simulation model's objective becomes the controlled variable of the control system.

  The process and actuators are both emulated by the computer simulation model (also referred to as the 'virtual factory').

- **Sensors**: The backward link between the physical process and software process is based on physical devices deployed onto the shop floor. These sensors (or clients) are responsible for continually monitoring the system. Manufacturing events are captured and communicated back to the event filter, whereas state information is communicated directly to a state store on the scheduling engine, this state buffer is used initialise the simulation model at compile time. Since the scheduling engine is only triggered when the rescheduling policy permits it, state information is buffered and only drawn upon when a rescheduling point is reached.

- **Comparator**: A comparator continually observes the difference between the selected scenario's estimated performance measure with that of the actual realisation of the system. Deviation of the real system beyond a specified tolerance results in the firing of a *deviation* event which, if permissible by the rescheduling policy, invokes the scheduling engine. This component is often referred to as an 'auto-validator' since it validates that the real system has responded appropriately to the control law.

  Due to the complexity of the auto-validator component and the time constraints of this thesis, the auto-validator was only studied theoretically, however its implementation was set aside for future research (see Section 9.2).

Once the preliminary literature study of reactive scheduling systems was complete, the basic characteristics and principles documented, and the purpose and relationship of the high level components understood, the design on such a system could begin. The next section deals with the inception phase of the project.

2.4 Project inception

It was decided at the outset of the project that an industry partner should be found to serve as the foundation for the project. This would ensure that the project had a basis in reality, and would expose any careless assumptions that would otherwise have been made. With reference to the characteristics and basic principles of an RSS discussed above, a set of high-level business requirements were drafted to define *what* the RSS should do.

Described a high level of abstraction, these requirements were, first, intended to assist in identifying an industry partner with suitable manufacturing characteristics, and secondly, to serve as the base requirements from which more specific requirements could be derived once an industry partner had been selected.

2.4.1 Business requirements for the simulation-based RSS

The business requirements identified were:

1. A simulation model of the manufacturing system (Section 2.4.2).
2. A means of capturing internal and external manufacturing events (disturbances) (Section 2.4.3).
3. A data model of the manufacturing system in which the configuration and state of the system could be stored (Section 2.4.5).
4. An application capable of receiving manufacturing disturbances, invoking the simulation model, interpreting the simulation outputs, and generating control instructions (Section 2.4.4).
5. A means of issuing control instructions back to the manufacturing system (Section 2.4.6).

Each requirement listed above is broken down into a second, more detailed, set of requirements in the sections below.

2.4.2 Simulation model of the manufacturing system

The core component of a simulation-based RSS is the simulation model itself. As essentially a virtual representation of the manufacturing system, the simulation model is critical to the scheduling ability of the system. The simulation model must:

1. Adequately represent the real system.
2. Expose an interface to allow control by the scheduling application,
3. Initialise off a snapshot of the manufacturing system, i.e. prior to replication running,
4. Estimate key scheduling performance measures and the occurrence of specific events, e.g. job completion times.

Details on the development of the simulation model are provided in **Chapter 3**.

2.4.3 Data acquisition platform

The operation of the simulation-based RSS is predicated on real-time state information being made available from the factory floor. A means of capturing the current state of the manufacturing system in sufficient detail, and passing the state to the reactive scheduling application is required. Specifically, the data acquisition platform must be able to perceive distinct physical events that are relevant to the goal of the system. The two event classes are:

1. Internal disturbances (e.g., WIP movement), and
2. External disturbances (e.g., new orders, order priority changes).

Further details on the requirements, design, and development of data acquisition components are presented in **Chapters 5** and **6**.

2.4.4 Reactive scheduling application

Management of the system and interfacing with the simulation model requires a reactive scheduling application to be developed. This application assumes the functions of the event filter, scheduling optimisation engine, and controller. The application should:

1. Manage system configuration,
2. Update the snapshot of the shop floor as new disturbances arise,

3. Provide an interface for the data acquisition and data visualisation components,
4. Process incoming events in accordance with the rescheduling policy,
5. Host the simulation model of the shop floor,
6. Invoke the simulation experiment and interpret the simulation model's output, and
7. Convert the model's output into meaningful control instructions.

Further details on the requirements, design, and development of the reactive scheduling application are provided in **Chapter 7**.

2.4.5 Database to support the reactive scheduling application

A database able to support the reactive scheduling application should be created. This data database must be capable of storing:

1. Configuration information necessary for running the RSS, and
2. State information needed for the initialisation of the simulation model.
3. Output generated by the reactive scheduling application.

The database is described in more detail in **Chapters 3** and **7**

2.4.6 Decision support platform

Embedding the selected control law requires that the manufacturing system be provided with:

1. Schedule control instructions capable of being executed by the actuators of the system, and optionally, to
2. Provide managers with meta-data resulting from the experimentation with the system (e.g., estimated completion time of entities)

These requirements are met in **Chapter 6**

2.5 Concluding remarks on Chapter 2

In this chapter the field of real-time stochastic scheduling was introduced, the characteristics of a reactive scheduling system were presented, and a brief overview of simulation-based reactive scheduling was given. The chapter concluded with a list of generic requirements set out to assist in identifying an industry partner with characteristics suited to an RSS. The following chapter introduces the selected industry partner and details the construction of the simulation model.

Chapter 3

Simulation model of a pressure gauge assembly operation

Simulation has long been established as a useful method of gaining insight into complex problems. The methods of differential calculus, probability theory, and other analytical techniques are seldom able to address the intricacies found in real-world systems. In such cases numerical, computer-based simulation can be used to adequately model the system's behaviour over time (Banks *et al.*, 2004). Formally, computer simulation is "the imitation of the operation of a system and its internal processes, over time and in appropriate detail to draw conclusions about the system's behaviour" (Law & Kelton, 2000). By constructing a quantitative model, often including stochastic elements, the effect of changing the parameters of the real system upon the responses of the system can be estimated by changing the parameters of the model. Thus insight into the real system can be obtained through experimentation with the simulation model.

3.1 Simulation overview

Discrete-event simulation is a tool for modelling dynamic systems, i.e., systems that change over time. The models of such system can be understood as compromising of a network of activities and delays—the arrangement of which defines the model's structure. Entities, objects of interest (e.g. customers) flow through the system via a particular sequence of activities and delays before exiting. The duration of these activities may be fixed or obey some random variable. In contrast the duration of the delays (e.g. the time entities spend in queues) is unknown, and arises from the system conditions. Resources (e.g. operators) provide services to entities such as facilitating entity movement, or to capacitate activities. At any one point in time the state of the system is defined by a

collection of variables. These variables describe what is happening in the system and change only at certain well defined points called events Banks *et al.* (2004). Such events govern the state progression of the model through simulated time. Thus, by observing the states and variables of the model, conclusions can be drawn about its operation through statistical analysis.

3.1.1 Simulation as an input-output transformation

Once a simulation model has been constructed, generally with the aid of computer software, the model can be used for experimentation. Figure 3.1 presents a conceptual view of the (simplified) statistical underpinning of simulation specifically where the comparison of k scenarios is of importance. A brief description of the figure is provided below.

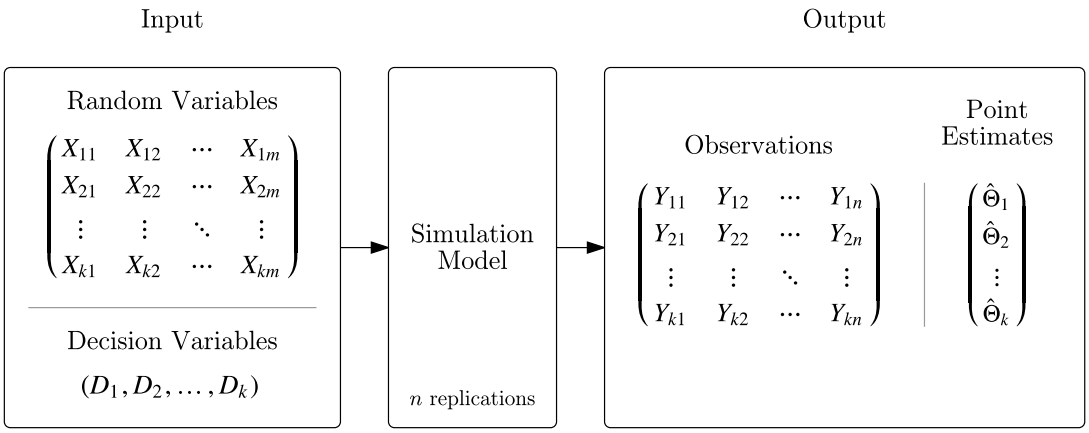


Figure 3.1: Simulation model as an input-output transformation.

A simulation model is built to approximate a real-world system with the aim of estimating some performance measure. Stochastic elements (denoted by X_{km}, where k denotes the number of scenarios, and m the number of random variables), are modelled either by drawing from real data or from statistical distributions. Each of the k constructed scenarios represents a particular configuration (or design) of the system; this configuration is defined by the set of selected decision variables (denoted by D_k)[1], and the structure of the model. Scenarios are run for n replications[2], where each replication

[1] Figure 3.1 depicts only a single decision variable, D_k, for each scenario k.

[2] Certain statistical procedures may vary the required number of replications per scenario, e.g. (Kim & Nelson, 2001).

outputs an observation Y, for each of the performance measure[1]. The distribution of these outputs is assumed to be normal[2] and may be characterised by a point $\hat{\Theta}_k$ and interval estimate. Comparison of the overlap of these confidence intervals (via a some method incorporating statical significance testing) results in the designation of one scenario as the 'best', and therefore the decision variables which defined the scenario's configuration may be considered to be the best. These decision variables may then be implemented in the real system with confidence.

An important aspect of Figure 3.1 is the role the simulation model plays as a function which transforms variable inputs into outputs. This principle forms the foundation of simulation experimentation. By inputting different values for the decision variables, different response values are obtained due to the transformation. For example, in a simple manufacturing system the throughput rate (response) of the system may be a function of the number of workers in the system (a decision variable). Building a simulation model gives the analyst access to the transformation function and a way to explore the impact the transformation has on the input variables. Thus, an attempt can be made, manually or by algorithm, at selecting input variables which improve the system with respect to one or more of its objectives.

Equation 3.1 expresses the principle of using a simulation model as a mapping from the decision space to the solution space,

$$(X, D) \xrightarrow{f} \hat{\Theta} \tag{3.1}$$

where X denotes the uncontrollable random variables, D denotes the controllable decision variables, f, the transformation due to the structure and behaviour of the simulation model, and $\hat{\Theta}$, the estimated performance measure.

3.1.2 Advantages of simulation

The advantages and disadvantages of simulation are well documented by Banks *et al.* (2004) and Kelton *et al.* (2010), one advantage regarding the building of simulation models using graphical software is worth expanding on. Simulation software—specifically modern packages with object-based model building interfaces—may enhance the modeller's understanding of the system and speed up the model building process. By interacting with a graphical representation of the system, rather than a code-base, the modeller is able to immediately obtain visual feedback on any changes made to the

[1] Figure 3.1 indicates only a single performance measure, Y.
[2] As a result of the central limit theorem.

model. That is, entity movements, resource interactions, queues states, and logical inconsistencies are mostly observable, and therefore immediately correctable. This high-level view frees the modeller from having to corroborate the effect of every change with output data—as would be needed when interacting with a code-based simulation or programmed analytical model. In summary, this build-run-observe-edit cycle, albeit at a high level, assists in speeding up the model building and verification process.

When faced with solving a problem in dynamic environments, specifically those that contain stochastic elements or exhibit intricate behaviour, analytical techniques often become too complex or fall short in adequately describing the system. This is particularly evident within manufacturing systems which are know to contain such complex behaviour. Analytical solutions are therefore difficult to derive without overly simplifying the problem (Banks *et al.*, 2004). Simulation is a modern and popular choice with which to address such problems because of its ability easily model and analyse—comparatively speaking—the stochastic behaviour and complex logic of real world problems.

3.2 Simulation as a scheduling technique

It is well know that majority of scheduling problems are NP-hard problems (French, 1982). That is, optimal solutions to these problems have not yet be found in polynomial time using existing analytical algorithms. Because of this, numerical simulation methods are also often chosen over exact or even approximation algorithms as they are able to yield 'good' results in reasonable time. Thus, the choice of simulation lowers the computational requirement—but comes at the expense of optimality. Nevertheless, for practical applications such as scheduling, non-optimal simulation results are still considered to be entirely satisfactory (Suwa & Sandoh, 2012). In light of this, the use of simulation as a scheduling technique is considered to be a practicable choice for a reactive scheduling system (RSS).

Basnet (2011) describes two general categories of problems that apply to scheduling of manufacturing systems. The first, involves intelligently selecting the set of orders to be released in the next planning horizon; the second, concerns the scheduling of orders within the current planning horizon. RSSs focus mainly on the second problem, which can be restated as: the problem of allocating resources to tasks to time-slots, with the goal of optimising one or more scheduling objectives (Błażewicz *et al.*, 2007). A common approach when using simulation to optimise a schedule or schedule control law, is to

compare the effect of adjusting the logic by which tasks are selected at workstations, i.e. comparing dispatching rules.

3.2.1 Dispatching rules as decision variables

Dispatching rules (selection disciplines) are a scheduling heuristic which priorities the tasks waiting to be processed at a workstation, i.e. those task in the input buffer. The rule is implemented at the instant the processor becomes idle, whereafter the task with the highest priority is dispatched to the processor (Pinedo, 2012)[1]. As noted by Kim & Kim (1994), when dispatching rules are employed, the act of scheduling resolves to the determination of the input sequence of tasks at each processor. Since dispatching rules are a heuristic method, the global optimality of the whole schedule over time cannot be guaranteed (Suwa & Sandoh, 2012). Nevertheless, Pinedo (2012) states that dispatching rule have been found to perform reasonable well with for respect to single objectives (e.g. flowtime) in real world scenarios.

3.2.1.1 Classification

Dispatching rules are commonly classified along two dimensions, local/global and static/dynamic[2] (Framinan *et al.*, 2014). To illustrate, let ϕ_i denote the list of eligible tasks at each workstation p_i, and t_d denote the time at which dispatching occurs. The classification is as follows:

1. Local/Global: This refers to the origin of the data that is considered when applying the dispatching rule. Local rules use only data from workstation p_i or list ϕ_i, where ϕ_i contains only those tasks residing at workstation p_i. Global rules make use of data originating from multiple workstations and queues, e.g. shortest queue at next workstation.

2. Static/Dynamic: Static dispatching rules apply the same priority to tasks irrespective the contents of the ϕ_i or the state of the schedule. Dynamic rules are dependent on ϕ_i, and attempt to take into account the state of the schedule prior to t_d.

In the next section an example of two local static dispatching rules and their effect on a schedule performance measure is illustrated.

[1] Dispatching rules apply only to eligible workstation input buffers, another related problem is *machine loading*, which is concerned with the rules for dispatching tasks *to* workstations. The reader is directed to (Pinedo, 2012) for more information.

[2] Independent of the static and dynamic classification of a rescheduling environment presented in Section 2.2).

3.2.1.2 Elementary example

A set of five tasks, $T_1, T_2, \ldots, T_5$, are to be processed on a set of three processors, $p_1, p_2,$ and, p_3. Each task T_i requires a single operation O_{ij} to be performed at each processor p_j. Processing times (shown in Table A.1) are assumed to be deterministic, idle time between operations is allowed, and operations must be processed beginning with processor p_1, and ending with p_3.

Two instances of the problem were created, each obeying a distinct dispatching rule. In the first problem, the first in first out (FIFO) rule was applied. In the second, the shortest processing time first (SPT) rule was applied. The schedules that resulted from the execution of the operations in accordance with the dispatching rule are shown in Figures 3.2 and Figure 3.3.

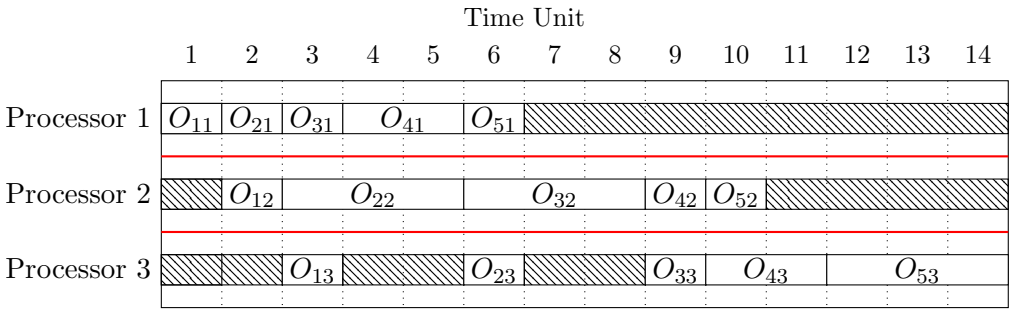


Figure 3.2: Dispatching rule: First in first out (FIFO).

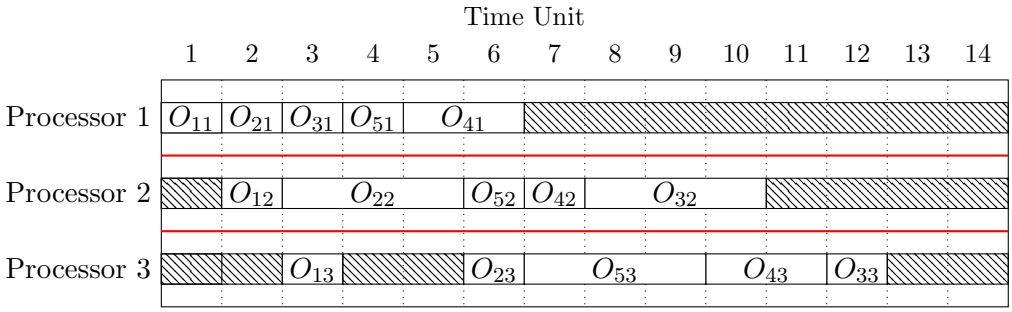


Figure 3.3: Dispatching rule: Shortest processing time first (SPT).

Figure 3.2 shows that when the dispatching rule FIFO is followed, the makespan manifested as fourteen time units. In contrast, when SPT is used to arrange the input buffers, as shown in Figure 3.3, the resulting makespan is twelve units. Thus, by simply

altering the ordering of the input buffer, a reduction of two time units was achieved. As a real-world analogy, had the operators of each workstation followed the control law of selecting the task with the shortest processing time, instead of the selecting task by order of its arrival, a total of two hours/days/weeks, for instance, could have been saved. The illustrated problem is, granted, an elementary one, however such improvements (as many authors have noted (Kim & Kim, 1994; Monch & Zimmermann, 2004; Pinedo, 2012)) have been observed in real manufacturing systems, making it a worthy and widely used technique.

Now that dispatching rules have been presented, the basic principles of an RSS, given in Section 2.3.2, may be expanded on.

3.2.2 Simulation-based reactive scheduling using dispatching rules

Building an RSS incorporating dispatching rules requires a number of supporting elements. Certain elements interface directly with the simulation model and important for understanding the role of the simulation model:

- Simulation experiment, within which scenarios are set up and replications run,
- Simulation optimisation engine (SOE), responsible for experiment execution and results interpretation, and
- Reactive scheduling application (RSA) responsible for the management of the SOE and interfacing with the shop floor.

Figure 3.4 shows the logical function of a simulation-based RSS. Manufacturing events (E_t) are continually communicated to a persistent store holding the state of the shop floor. As rescheduling points arise in real-time, the simulation model is initialised with a data snapshot of the factory floor (S_t). Control is then handed over to the SOE which calls the simulation experiment, requesting a dispatching rule be issued to each of the k simulation scenarios. After completion of the experiment, the SOE interprets the experiment's results and selects the best dispatching rule (D_k) according to some procedure. This rule is returned to the shop floor for use in the next period, and the cycle is repeated.

With reference to the example of Section 3.2.1.2, the data snapshot would contain the set of tasks awaiting to be processed, events could be triggered, for example, by the arrival or departure of tasks at processors, the dispatching rule selected would translate

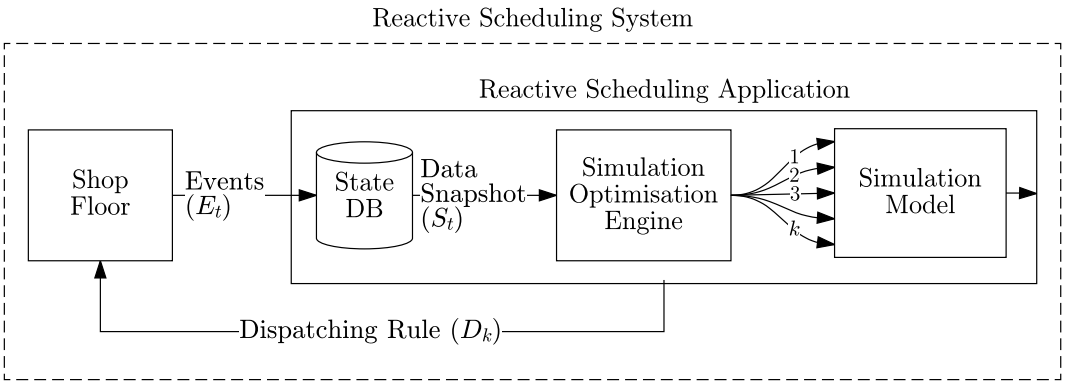


Figure 3.4: Simulation-based reactive scheduling using dispatching rules.

in to a control law and returned to the shop floor.

Naturally, integration between the simulation model and the SOE was critical to the success of the project. The model needed to expose an interface for the SOE to call into, allowing:

1. Control the parent experiment,
2. Initialisation of the model with the state of the factory floor at the start of each replication run,
3. Access the results generated by the experiment, i.e. performance measures and events.

At this point, an overview of the basic principles of reactive scheduling (Section 2.3) has been presented; a feedback control analogy was given (Section 2.3.3), in which the shop floor was compared to a process who's output is to be controlled; numerical simulation (Section 3.2) was presented as a valid scheduling technique; and the use of dispatching rules as a scheduling method (Section 3.2.1) was examined and an example to demonstrate their value was given. In this section the role of the RSA and simulation experiment were explored. With this knowledge and understanding, the development of the simulation model could begin.

3.3 Development of the simulation model

As stated in **Chapter 2**, a real world manufacturing problem needed to be found to serve as a test-bed for development of the RSS. Thus, a manufacturing system conducive to both a simulation study and reactive scheduling via dispatching rules needed to be found. With guidance from the literature review thus far, a set of criteria were constructed with which to the evaluate potential industry partners:

1. Manufacturing system: flexible, job, or flow.
2. Minimal automation.
3. Operator dependent manufacturing.
4. Suitable for computer simulation.
5. Commitment from management.

A after compiling a list of candidate industry partners, a pressure gauge manufacturer based in the Western Cape province of South Africa, was chosen. The partner's business focussed on the manual assembly of pressure gauges in a job shop environment. The objective of the study shifted to modelling the gauge assembly process in sufficient detail for use in the RSS.

3.3.1 Description of the gauge assembly process

The gauge assembly operation was conducted on shop floor with a footprint of approximately 1 600 m^2, the facility contained 38 workstations, 19 operators and one floor manger. With roughly 41 000 gauges assembled annually, the facility may be classed as a high production facility (Groover, 2008). *Utility* and *process* gauges (the two main gauge types considered) accounted for 80% of yearly volume with a volume split of 80% and 20% respectively[1]. The facility was operational 4 and half days per week (see Table A.2), though operational hours are extended during periods of high demand. Assembly was performed manually by operators at fixed workstations. All operations required a single operator (selected from a set of capable operators) to be present at the associated workstation. Work-in-process build up in workstation buffers was common.

Orders arrived at the system boundary where they were handled by production planning and control. For each new order a *tasksheet* (A4 page) was drawn up which

[1]Based on production data from 2012.

captured the attributes of the order, such as the: priority, due date, gauge quantity, gauge type, etc. At this point the order was referred to as a job, and each job was assigned a tasksheet. Jobs with a gauge quantity exceeding 50 units were separated into sub-batches (tasks). Jobs with fewer than 50 gauges were associated with a single task. The tasks inherited all the attributes of the parent job, e.g. due date, and received a tasksheet of their own (if the job was split up). In summary, arriving orders were captured as jobs, and were then split up into a single or multiple tasks, where each task comprised a batch of gauges (typically between five and thirty).

After the creation of a task, material requirements were checked. If all required components were in stock and the dial design template existed, the task was considered to be ready for assembly and was added to the *release* queue. At this point the task was ready for processing and was placed alongside its associated tasksheet in a store room ready for component picking. If the required components were not in stock or if the dial design was still to be created, the task was added to an *on-hold* queue where it remained until the requirements were fulfilled.

The assembly operation focused on two main gauges type: *utility* and *process*. Process gauges were of a better build quality, self-repairable and were calibrated to be accurate to within 1% of the true pressure value. Utility gauges were cheaper, not as easily repaired, and were calibrated to within only 1.6% of the true value. Numerous sub-categories exist within these two types depending on a number of factors such as the pressure range, case material, thread size, tube type, calibration certificate, dial colours, gauge size, etc. Additionally each gauge may be customised to suit the customer's application, e.g., high vibration environments require gauges to be liquid filled to steady the indicator needle. This manifested in a wide variety of gauges and corresponding assembly sequences, typically though, sequences consisted of approximately eighteen operations.

As tasks (and the associated gauges) moved through the assembly operation, operators were responsible for determining the order in which to process them. Influencing their decision making was the task's assigned priority, and secondly, the order in which it arrived at a workstation. Expedited jobs were common as, in some real-world use cases, gauges play a crucial role in monitoring process health, e.g., gas cracking, thus orders are expected to be processed quickly to minimise downtime.

After moving through roughly seventeen operations, the final operation for all gauge sequences was packaging, where the gauges were wrapped, packed in boxes, and readied for shipment. Only once all the tasks associated with a parent job had reached this step

was the job green-lit for dispatch. At this point the job was considered to have exited the system boundary. In summary, the assembly process may be characterised by:

- a diverse product mix,
- batch processing of gauges,
- multiple assembly sequences,
- machine-worker assembly operations, and
- potential for dispatching rule implementation.

The assembly operation's characteristics described above were conducive first, to simulation study, and secondly, to the integration into a simulation-based RSS. The dynamic nature of the assembly process, system complexity in the form of diverse job routings, floating and fixed secondary resources (operators), stochastic elements such as random gauge processing times, and workstation input buffers with capacity to hold more than one WIP item lead to the selection of the gauge assembly process as the test bed for this thesis.

3.3.2 Research approach

A traditional systematic approach was adopted in designing and developing the simulation model. The Steps of a Simulation Study, described by Banks (1998), were used as a reference. Figure A.1 of Appendix A presents these steps in the form of a flow diagram. The structured approach proved helpful as the complexity of the assembly process presented numerous challenges during model translation.

3.3.3 Data acquisition

To gain domain knowledge of the assembly operation at the industry partner, numerous site visits were conducted. Activities performed on these visits included interviews with management, production and control staff, floor managers, and workstation operators. Additionally, numerous direct observations were made of the assembly process. Remarkably little documentation existed at project inception, and so a period of some months was spent documenting the gauge assembly process. The data collection and documentation phase was executed in five steps which are described below.

1. **Floor Plan:** The documentation phase began with the creation of a floor plan of the facility, shown in Figure A.10, which included the physical structure of the building and workstation locations. The floor plan was used as a blueprint for the building the simulation model, and assisted in mapping operator paths and the sequence of workstation visits made by each operator.

2. **Operation sequences:** Process flow diagrams were created for each of the two main types of gauges: process and utility, these diagrams are shown in Figures A.4, A.5, and A.6.

3. **Candidate operators:** A set of candidate operators for each workstation was compiled. The candidate set includes only the operators that are trained to perform the operations associated with the workstation.

4. **Processing times:** After the operational sequences were finalised, eleven data collection forms (designed to capture the most frequently occurring sequences) were created. The purpose of these forms was to collect timestamp data on the movement of tasks through the system. Operators were asked to fill in a timestamp value at each instant that a task was: *received,*, *stated*, *completed*, and *dispatched*. Data collection sheets were attached to task entities and accompanied them through the factory. These data were analysed in order to determine a suitable random variable to represent the processing time of each workstation. Cross Ref!

5. **WIP snapshots:** To obtain suitable test data, the state of the factory floor was captured over 5 successive days. Snapshot forms were issued to each processor and operators asked to fill in each form before the day's processing began. This daily 'snapshot' was used to build up a picture of the both the *quantity* and *distribution* of WIP typically found in the system. A summary of the observations is shown in Table B.1. The distribution of the WIP across the factory is not shown here, but is captured in the test data off which the simulation model initialises each run.

The data collection phase culminated in the creation of a concept model which is described in the next section.

3.3.4 Concept model

A concept model was created to describe the structure and logical relationships of the components of the system (Banks, 1998). The boundary of the system was defined as

encompassing the factory floor and all operations conducted within it and related specifically to the assembly of pressure gauges; this included all assembly related components, workstations, operators, materials, and equipment.

Only the essential features of the real-world system such as, job releases, job completion, sub-batching, dial and movement material creation, WIP handling, operator-processor interactions, etc., were extracted in order to develop the concept model which is shown in Figure 3.5. This model, apart from describing the logical relationships between the components and structure of the system, indicates the input and output data required and produced. The concept model was key in understanding system and in development of the computerised model.

3.3.5 Assumptions

Certain assumption were necessary to scope the model, these are as follows:

1. With the exception of dial faces and movements, all components necessary for the assembly of a gauge are in stock. This implies that the assembly of a gauge may not be interrupted due to a stock-out.

2. Operators that perform processing on a task are responsible for the transportation of the task to the next workstation. This is generally, but not exclusively, followed on the shop floor.

3. Rework is not modelled for the purposes of this thesis. The added complexity of modelling rework would not contribute to the goal of this thesis.

4. Machine failure is not modelled. The added complexity of modelling rework would not contribute to the goal of this thesis.

5. Instantiated WIP is assumed to have both dial and movement components available regardless of the instantiation position in the assembly sequence.

6. Preemption is not allowed. Operations must be completed before the operator is released.

7. Inter-workstation operator travel times are assumed to be constant.

These assumptions would need to be addressed if the model is to form part of a real-world RSS.

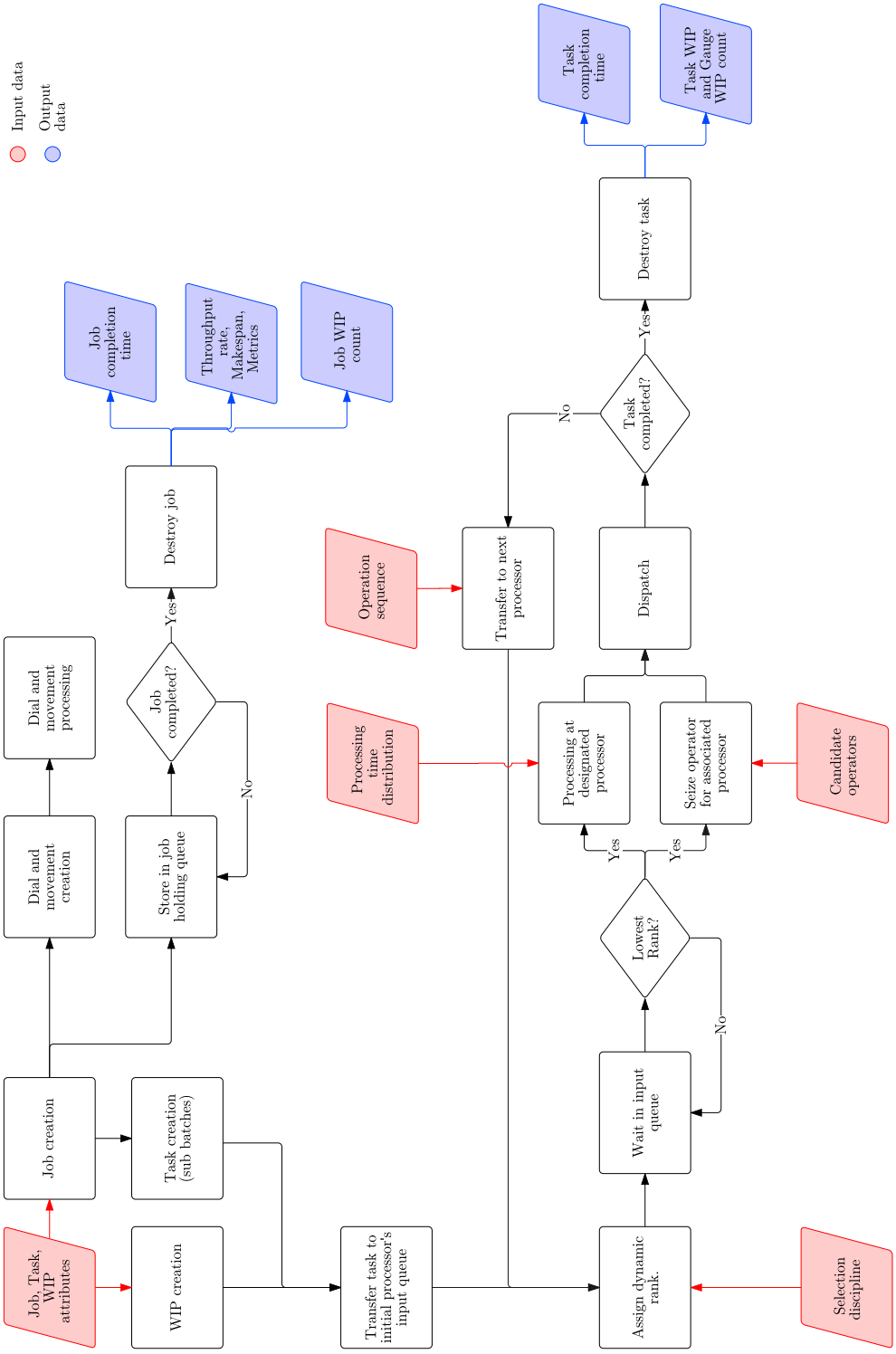


Figure 3.5: Concept model of the gauge assembly process.

3.3.6 Features of the model

A few of the important features of the model are highlighted below.

- **Assembly streams:** Three assembly streams operate in the model. The main stream is responsible for the assembly of the gauges, whilst two parallel streams produce the components necessary for the main assembly, namely the dial face and movement mechanism. The production of theses two components was modelled as their output is critical for the main sequence and because their operators are shared among workstaions in the main assembly process.
- **Entities in the model:** Five entities move through the system: jobs, tasks (pallets), gauges, movements, and dials.
- **Operators:** Due to the level of involvement of the operators it was necessary to model their interaction with the tasks and their movements between workstations. A one-to-one mapping was created between operators within the factory and those in the simulation model. Workstations were programmed to seize only those operators available in its candidate operator list.
- **Consecutive sequences:** Certain operations needed to be performed consecutively and by the same operator. This was achieved by 'locking' operators to tasks until the sequence had ended. Since the model was built to scale the travel time between workstations was representative of the real system.
- **WIP instantiation:** The model was programmed to allow WIP items to be instantiated in the input or output buffer of any workstation. This non-trivial feature allowed the state of the real system to be initialised in the simulation model, essentially creating a one-to-one mapping between entities on the shop floor and entities in the simulation model.
- **Off-shift:** Off-shift periods are modelled. That is, during tea, lunch and after hours periods the facility does not operate. This was critical since the expected completion time of jobs was a required output. Time variant performance criteria, e.g. throughput, were adjusted accordingly.

Figure 3.6 shows the simulation model of the assembly operation at the industry partner. Operators can be seen performing activities on tasks at workstations. Inter-workstation transfers are visible.

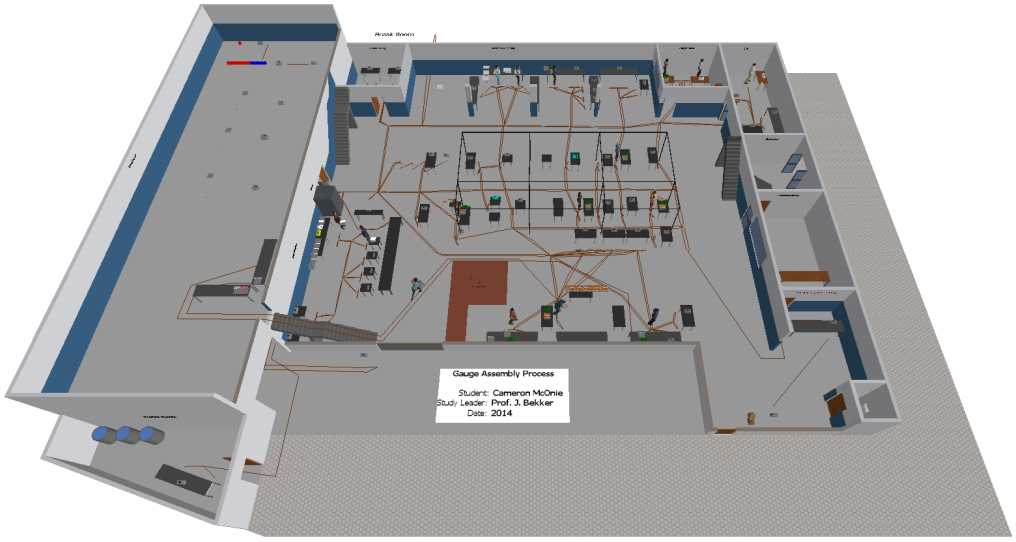


Figure 3.6: Simio® model of the industry partner.

3.3.7 Input Analysis

Estimates of the processing times of each gauge type was achieved by analysing the data gathered from the timestamp data collection forms (described in Section 3.3.3). These data were first processed to remove all off-shift intervals, i.e. periods in which the WIP item had commenced processing but had been interrupted due to an off-shift period. To characterise the data set a theoretical triangular distribution was fitted to the data of each workstation.

3.3.8 Output analysis

Estimating a simulation model's true performance measures is a statistical issue and must be given careful consideration (Banks *et al.*, 2004). Further, the use of simulation as a scheduling technique in an RSS imposes constrains on the type of output analysis that must be performed, as described below.

The simulation model is classified as a terminating system, since the model begins at T_0, and runs for the duration $[T_0, T_E]$, where E is an event that stops the simulation. Three important issues regarding output analysis are discussed below.

1. **Run length**: Since the rescheduling environment was static, i.e., a finite set of jobs were considered, the simulation model was constrained as a terminating system (as there were finite entities in the model). In such cases the scheduling problem must

be addressed as a series of static problems that are solved on a rolling-horizon basis (Kim & Kim, 1994). This is in contrast to off-line output analysis which is primarily concerned with estimating the steady state performance of the system.

Termination of the model was handled internally by the simulation model logic. Termination occurred at the time instant at which the last job exited the system—considered to be the final event, T_E.

2. **Warmup period**: The model was designed to observe a continuously shifting window (rolling horizon), where it was instantiated with the state of the real system at T_0, thus no warmup period was required as the model was started in valid and representative state. Furthermore, such a myopic model will never reach steady state over such short planning horizons, and therefore focus is only on the transient phase Banks (1998).

3. **Required replications**: Since k scenarios (each corresponding to a dispatching rule) needed to be compared and the best scenario selected (with respect to the chosen objective) a means of comparing each of the k systems was needed. The Kim Nelson ranking and selection procedure of Kim & Nelson (2001) was chosen since it guarantees, with probability $1 - \alpha$, that the best scenario will be chosen if it is at least greater than each of the other scenarios by a user specified amount, named the indifference zone. Further details of this procedure are described thoroughly in Chapter 7.

After the topic of output analysis was taken care of, attention was given to the control and measurement of the system.

3.4 Performance measurement and control

Establishing one scenario as best suited to the shop floor meant that one or more measures of performance had to be monitored within each scenario. By having a handle on the performance of each scenario, the efficacy of the decision variable input could be evaluated and compared.

A fluctuation in gauge demand prompts the industry partner to change objective. To accommodate this, functionality was added to allow a single objective to be chosen from the set of performance measures described above. A better approach would involve the optimisation of multiple objectives, though this is beyond the scope of this study and is offered as future research (see Section 9.2).

3.4.1 Schedule performance measures

To determine what to measure in the model, shop floor managers were interviewed and asked which measures were of interest to them. The selected performance measures and their calculation are described below.

Upon completion of each simulation scenario, the following performance measures were observed for each task T_j.

1. *Estimated* completion time $\hat{C}_j$ with *expected* completion time C_j.
2. *Estimated* flow time $\hat{F}_j = \hat{C}_j - r_j$ with *expected* flow time F_j.
3. *Estimated* lateness $\hat{L}_j = \max\{\hat{C}_j - d_j, 0\}$ with *expected* lateness L_j.
4. *Estimated* earliness $\hat{E}_j = \max\{d_j - \hat{C}_j, 0\}$ with *expected* earliness E_j.

where, r_j is the release date, and d_j the due date of task T_j.

In order to estimate the completion time of each job $\hat{C}_j$, as described above, the completion event of each associated task T_j in each replication needed to be monitored, the completion time of a job was then *estimated* by

$$\hat{C}_i = \frac{1}{p} \sum_{j=1}^{p} \hat{T}_j \tag{3.2}$$

where p the number of tasks the job is comprised of. Similar calculations were encoded into the simulation model for each of the performance measures.

As discussed in Section 3.3.1, each job is comprised of one or more tasks, where each task contained a batch of gauges. Since jobs were not shipped until *all* tasks associated with the job were completed, performance measures at the level of the job were required. These performance measures were calculated by aggregating the performance measures of the task(s) from which it was comprised. These metrics are described below, where the job count in each scheduling window is denoted by m. Equations for both the estimated measure of the sample population, and the expected value of the true population parameter are given.

Makespan

Makespan is defined as the total amount of time required to complete the set of jobs in the system, also know as the schedule length. This measure was found by selecting the maximum completion time from the set of jobs in the system, *estimated* by

$$\hat{C}_M = \max\{\hat{C}_i\} \tag{3.3}$$

with *expected* schedule length $C_M = \max\{C_i\}$.

Flow time

The average time jobs spent in the system was calculated by averaging the flow time of each job in the system, *estimated* by:

$$\bar{\hat{F}} = \frac{1}{m}\sum_{i=1}^{m}\hat{F}_i \tag{3.4}$$

with *expected* mean flow time $\bar{F} = \frac{1}{m}\sum_{j=1}^{m} F_i$.

Cumulative lateness

The average lateness measure gives an indication of how late jobs are by averaging the duration of time jobs spent in the system *after* their due date, *estimated* by

$$\hat{L} = \sum_{i=1}^{m}\hat{L}_i \tag{3.5}$$

Further, if $\hat{L}$ was found to be negative, the metric conveyed the cumulative earliness of the jobs. The *expected* cumulative lateness is given by $L = \sum_{i=1}^{m} L_i$.

Number of tardy jobs

Another measure of lateness is the number of tardy jobs, i.e. the count of jobs not completed by their due date, this measure is *estimated* by

$$\hat{U} = \sum_{i=1}^{m}\hat{U}_i, \text{where } \hat{U}_i = 1 \text{ if } \hat{C}_i > d_i, \text{ and } 0 \text{ otherwise.} \tag{3.6}$$

where d_i is the job due date, and where the *expected* number of tardy jobs is given by $U = \sum_{i=1}^{m} U_i$.

In addition to these performance measures, both the throughput rate and WIP level of gauges, tasks, and jobs were also encoded into the model.

3.4.2 Dispatching rules as model controls

The simulation model was built with a single configurable property, named *dispatchingRule*, meaning the dispatching rule could be set on a per-scenario basis. This property gave the simulation experiment control over the dispatching rule employed by the scenario. This fact allows the transformation expressed in Equation 3.1, to be modified to include the dispatching rule as the single decision variable:

$$(X, \text{Dispatching Rule}) \xrightarrow{f} \hat{\Theta} \tag{3.7}$$

making the transformation is a function of both the stochastic variables of the model and prescribed dispatching rule.

After reviewing the works of Błażewicz *et al.* (2007) and Pinedo (2012), seven local static dispatching rules were selected. Listed in Table 3.1, these rules were chosen due to their widespread use in industry today. Thought many more rules exist, these seven were deemed to be sufficient for the construction of an RSS.

3.4.3 Dynamic rank assignment

Each workstation was encoded with rule switching logic which observed the *dispatchingRule* property. Once the rule was determined, workstations evaluated a *dynamic rank* variable associated with each task currently residing in the workstation's input queue. This rank evaluation was triggered at any instant in which capacity became available at the workstation. The numeric value of a task's dynamic rank variable was programmed as a function of the dispatching rule of the parent scenario and certain attributes of the task.

Table 3.2 illustrates typical attribute values of a set of four tasks residing in a workstation's input queue. Indicated are: the task's release and due dates; the timestamp of the instant the task was received by the workstation; the number of gauges associated with the task; and finaly, the mean processing time in minutes.

Table 3.3 provides an example of the numeric rank applied to each task after dynamic ranking evaluation was performed. Importantly, this ranking is set to change at the instant that the workstation becomes available for processing. This process of dynamically ranking tasks in each workstation's input buffer is the central principle of using dispatching rules as a scheduling method.

Table 3.1: Dispatching rules used for model control.

Abrv.	Name	Description
EDD	*Earliest due date*	Priority equals the due date of the imminent task.
ERD	*Earliest release date*	Priority equals the release date of the imminent task.
LNG	*Least number of gauges*	Priority equals the task with the smallest gauge sub-batch size.
LPT	*Longest processing time*	Priority equals the inverse of the processing-time of the imminent operation.
FIFO	*First in first out*	Priority equals the sequential order-of-arrival of the tasks at the processor's input queue.
SIRO	*Service in random order*	Priority is assigned randomly to operations within the processors' input queue.
SPT	*Shortest processing time*	Priority equals processing-time of the imminent operation.

Table 3.2: Typical attribute data used for dynamic rank assignment.

Task Id	Release date (r_j)	Due date (d_j)	Received	Batch	Mean
1000	01/01/2014 10:00	05/01/2014 12:00	02/01/2014 08:30	8	1.92
2000	01/01/2014 09:35	02/01/2014 16:00	02/01/2014 08:02	25	6.00
3000	02/01/2014 08:35	03/01/2014 09:00	02/01/2014 09:41	100	24.00
4000	03/01/2014 09:50	04/01/2014 10:30	05/01/2014 13:00	16	3.84

Table 3.3: Example of dynamic ranking assignment results.

Processor: XYZ					DateTime: 2014/01/01 13:42		
Task Id	SPT	LPT	ERD	EDD	LNG	FIFO	SIRO
1000	1	4	2	3	1	2	0.72
2000	3	2	1	1	3	1	0.30
3000	4	1	3	2	4	3	0.82
4000	2	3	4	4	2	4	0.39

Two separate and isolated implementations of each dispatching rules were required. One within the simulation model, and the other within the dynamic scheduling application, whose purpose is to rank the tasks on the shop floor, once a dispatching rule has been determined to be statistically significantly better than the others.

3.4.4 Current scheduling procedure of the industry partner

The current scheduling strategy employed by the industry partner can be described as completely reactive, i.e., no baseline schedule is generated in advance. Operators responsible for processing tasks at workstations follow two simple rules:

1. If any task in the input buffer has a priority assigned to it (i.e. the order has been expedited) processed the task first,
2. else, select tasks according to their order of arrival at the input buffer (FIFO).

Both expedited orders and the dispatching rule FIFO were programmed into the model to serve as the control experiment.

3.5 Model verification and validation

Modelling of any system would not be complete without a systematic procedure for checking that the model is credible in both its functioning, and its output. Banks (1998) describes two techniques, verification and validation, which are useful for the credibility checking of a simulation model. In this section definitions of these two components are given and their application to the simulation model is described.

1. Verification: The process of reconciling behavioural differences between the simulation model and behaviour prescribed by the conceptual model. The goal is to ensure consistency between the conceptual and computer models.

2. Validation: The process of establishing whether the conceptual model does in fact represent the real system with respect to the objectives of the study.

Three verification and validation techniques were applied namely: model reasonableness, face validation, and evaluation by a subject matter expert.

3.5.1 Model reasonableness

Law & Kelton (2000) provide four verification and validation factors that, if approached correctly, contribute to model credibility. Essentially these factors involve making changes to the model input and observing the affect on the behaviour and response of the system. The factors are stated below and examples supporting them are provided.

- **Continuity:** Altering the input data of the model reflected in the output data of the model. Increasing the mean processing time of each workstation by a factor of ten resulted in proportional increase in estimated job completion times.
- **Degeneracy:** When the number of jobs released into the system was decreased the average WIP level decreased; as well as the total length of the schedule.
- **Absurd conditions:** When the model was run using test data, in which 80 jobs were set to be released into the system at the same instant with zero WIP, the model behaved in a predictable manner. WIP increased to its maximum immediately and gradually decreased over time until all jobs had completed. Noticeably more tasks occupied input and output queues. Utilisation of upstream processors was high.
- **Consistency:** Performance estimates were consistent when the model was run using the same dispatching rule.

These factors provide confidence in the credibility of the model.

3.5.2 Face validation

Face validation was used to ensure the model represented reality at a respectable level. This was performed using a combination of visual inspection and ad-hoc experimentation.

Table 3.4: Simulation model face validation criteria.

Function	Criteria	Validated
Jobs	Jobs released into the system at the instantiation datetime instant.	√
Tasks	Correct number of tasks created for each job in system.	√
	Operation sequence of the associated gauge is obeyed.	√
	Task awaits movement and dial material creation.	√
Gauge	Correct number of gauges batched and associated with a pallet.	√
	Correct gauge type associated with pallet.	√
Dial and movement	Correct number of dial and movement pallets created.	√
	Correct amount of dials and movements material created.	√
Operators	Processors seize only candidate operators.	√
	Operators transport current pallet forward.	√
	Certain operators obey uninterrupted processing and transport sequences.	√
	Operators are unavailable during off-shift periods.	√
	Operators obeyed workstation associations.	√
WIP	Correct number of WIP tasks instantiated.	√
	WIP tasks instantiated at the correct processor and in the correct queue.	√
	Do the WIP tasks	√
Dispatching rules	Tasks are selected in the correct order—based on their dynamic rank property for all dispatching rules.	√
Other	Off-shift hours excluded from available assembly time.	√
Input data	Relational table data match the corresponding database table data upon import (both manual and automatic).	*
	WIP data correspond to the latest WIP events	
Output data	Job expected completion times match the simulation clock when exiting.	√

3.5.3 Model evaluation by a subject matter expert

The simulation model was further validated by a subject matter expert (SME). The SME interviewed was intimately familiar with the system, having twenty years of experience in pressure gauge assembly operations. According to the SME, the model represented both the structure and behaviour of the system at a reasonable level. Job completion times appeared to fall within typical internal estimates. Operator movement and WIP level appeared similar. The SME concluded that the model was of a sufficient level of detail for the purposes of this project.

The combination of model reasonableness, face validation and SME evaluation ensured that the simulation model could serve as a credible test-bed for the simulation-based RSS developed in this thesis.

3.6 Concluding remarks on Chapter 3

This chapter began with a brief overview of simulation, followed by the introduction of dispatching rules and their use in simulation-based scheduling. The selection of an industry partner, a pressure gauge manufacturer, was discussed and a description of the operation was given. Further, the development of the simulation model was described, where after the selected performance measures were presented and implementation of dispatching rules discussed. The chapter concluded with verification and validation of the model.

This choice in modelling a real world system introduced a high level of complexity into the simulation model, and subsequently the reactive scheduling system built around it, however valuable insights into real challenges facing such a system were gained.

Chapter 4

Selection of a software life-cycle model

Software projects, as with all complex projects, must be executed within a formalised life-cycle model if they are to have any hope of success. These life-cycle models (or development methodologies) allow an individual or team to plan, structure, and control the process of creating a software product. Since the emergence of the software field, a variety of life-cycle models have been proposed, each attempting to address the inherent difficulty of running a successful software project.

In the absence of a formal software life-cycle model, software development tends to fall back onto the 'code-and-fix' model. This model is attractive because of its short lead time, minimal overhead, and quick turnaround—and in some cases may even be the preferred method (typically for small projects of under 200 lines or so). However, as complexity of the software product increases, drawbacks manifest. The lack of clear objectives, a well defined feature set, and a vague architecture, result in confusion around what the software should achieve, and how to achieve it. Furthermore, the assessment of project metrics, e.g. progress, performance, quality, risk, etc., is made more difficult. Consequently, no serious software project is undertaken without a formal life-cycle model to guide it. The selection of one for the development of the RSS was therefore imperative, and is the subject of this chapter.

4.1 Overview of software life-cycle models

Simply put, a software life cycle model is the series of steps, or structured set of activities, that are performed while the software product is being developed and maintained

Schach (2010). Common to all life-cycle models in literature are the fundamental activities of: requirements gathering, analysis, development, verification, implementation and maintenance. Furthermore, two broad categories of life-cycle models appear: *linear* and *iterative*. In the former, a predefined linear sequence of steps is followed until the software product is complete, whereas in the later, all or a portion of the sequence of steps are followed iteratively, i.e. the life-cycle contains cyclic elements. Within these two categories, various life-cycle models have been developed.

This section provides a brief overview of three popular life-cycle models: waterfall, iterative and incremental, and agile with the purpose of giving the reader context around the selection of one for use in this thesis.

4.1.1 Waterfall

As one of the first life-cycle models in the field, the waterfall model (Royce, 1970) proposed that the fundamental process activities of: requirements, design, implementation, verification, and maintenance be considered to be discrete phases (see Figure 4.1) (Sommerville, 2011). Each phase is executed in downward flowing sequence, and must be completed (and typically 'signed off') before the next phase may begin. As a consequence, the model is popular with proponents of 'big-design-up-front', since all analysis and design must be completed before development begins. This systematic approach has yielded many working systems and possesses a number of favourable traits such as: easy cost tacking, simple block-like project scheduling, and clearly defined roles and responsibilities.

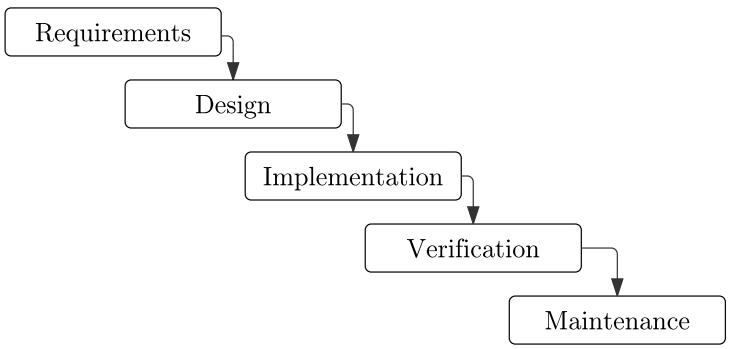


Figure 4.1: Pure waterfall life-cycle model.

Drawbacks of the model include: 1) little to no feedback as information is passed downstream only. 2) designers of the system are required to be intimately familiar with the technology that will be used to develop the system, i.e., limitations and constraints imposed by the underlying technology must be foreseen during the requirements and design phases. 3) as system complexity rises, it becomes increasingly difficult to resolve all business requirements into a single design. The waterfall model is therefore often termed "idealised software development" (Schach, 2010).

To address the aforementioned drawbacks many improvements and custom implementations of the waterfall model have been proposed, including Royce's own modified waterfall model (Royce, 1970). These, however, will not be explored since newer models which circumvent these issues altogether have since been proposed.

4.1.2 Iterative and incremental

The iterative and incremental life-cycle model (Jacobson *et al.*, 1999), generalised in Figure 4.2, proposes a cyclic model in which the product is evolved through series of versions (or increments) and where each version adds functionality (Sommerville, 2011). As an evolution of the spiral model (Boehm, 1988), the model caries forward the idea of identifying risks early in an iteration cycle; issues can therefore be addressed upfront as opposed to at the end of the project. In essence, iterative models spread decision making throughout the life of the project with the assumption that knowledge gained in one iteration will improve decision making in iterations that follow.

A number of advantages arise from iterating over the software creation process:

1. As each increment is released, the customer is provided with a working, albeit limited, product. Thus, value is continually added to the customer as the product increments toward the end deliverable.

2. User feedback is received early and continuously throughout the project. In doing so, possible shortcomings or improvements may be identified and incorporated into the work log.

3. The tight coupling between the design, development, and testing phases promotes team interaction and offers the opportunity to correct problems in the requirements and analysis phases. Such problems would otherwise have gone unnoticed until surprises or limitation arose later in the project. This is specifically relevant for exploratory projects in which the technology stack is still unproven.

Drawbacks of the iterative approach include:

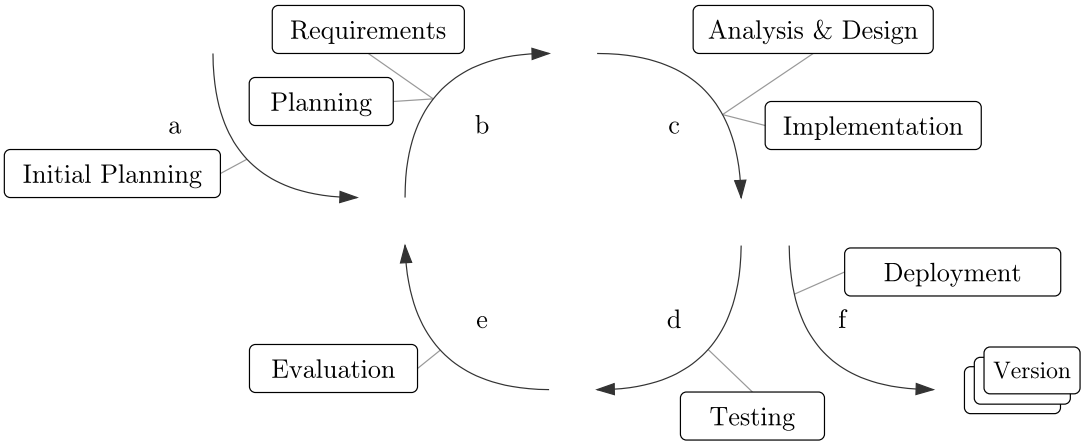


Figure 4.2: Iterative and incremental life-cycle model.

1. Difficulty in scheduling resources as the project may be in multiple stages at once.
2. A lack of visibility into the process.
3. Time required for frequent document revision and architecture updates.
4. Time needed to refactor code[1].

Despite these drawbacks, the iterative model is widely and successfully employed in industry Schach (2010). A more recent branch of the iterative model, is the agile modelling process. Since agile has seen significant growth in the past decade, and developed into a discipline of its own, it presented independently.

4.1.3 Agile

The agile model presents a formal system in which a series of increments (or sprints) are executed until the full feature set is delivered (see Figure 4.3). Two common implementations are Scrum (Schwaber & Beedle, 2001) and Extreme programming (Beck, 2000). The agile process is guided by four core principles: short releases, 40-hour workweek, on-site customer, and people not process through pair programming (Kendall & Kendall,

[1] Without a big-think-up-front, in which all software features and their relationships are considered, incremental feature addition may result in code duplication giving rise to software complexity and code inefficiency. Refactoring, is therefore key to keeping code simple and efficient. Refactoring, however, is time and design intensive, and may force undesirable interface changes.

2011). These principles are intended to support the software effort. For example, short releases (which are typically time-boxed (Jalote *et al.*, 2004), i.e. features are scoped or de-scoped in order to match the time-frame of a release) set the project cadence. Additionally, on-site client representatives may become fully integrated into the development team, thus client requirements and product verification may then be arranged into the daily workflow, reducing the risk of delivering a product not suited to the client's needs.

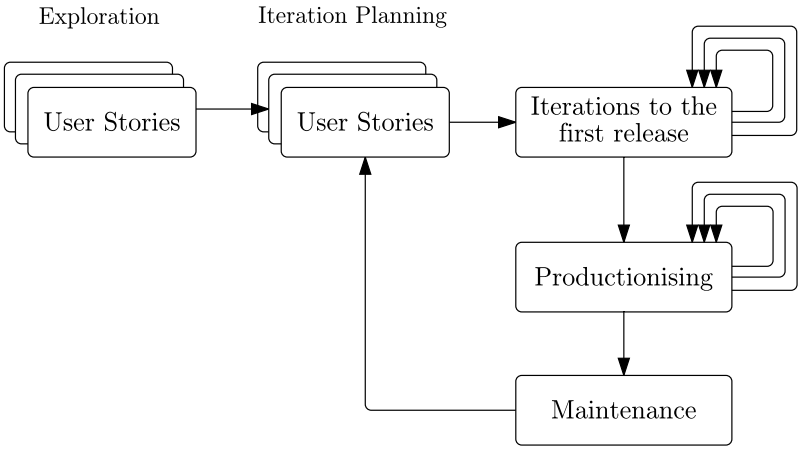


Figure 4.3: Agile life-cycle model (Kendall & Kendall, 2011).

The agile model shares many of the same disadvantages as the aforementioned iterative and incremental model, though some are more pronounced. Tight client integration may result in excessive client training for each project, resource scheduling is difficult when a shared resources pool supports multiple projects, and refactoring remains a challenge, especially toward the end of a project.

4.2 The unified process—an iterative life-cycle model

After the literature review of the three software life-cycle models was complete, each models' advantages and disadvantages were analysed, and their suitability to the development of a simulation-based RSS was considered. Important consideration was given to the following risks:

1. Requirements may change as the project unfolds.
2. Host of unfamiliar technologies.

3. Multiple programming languages and software frameworks.

4. Dependency on an API from third-party simulation software.

The iterative and incremental model was chosen primarily for its property of allowing knowledge from one iteration to feed into the next. It was realised that an attempt at a complete up-front specification of the RSS was not possible as an insufficient understanding of the components and their interactions was known at the outset of the thesis.

Agile too, offered to address of these risks, however the methodologies of scrum and extreme programming are specifically geared toward *team* development and thus would be difficult to apply in the individual setting of the research environment.

4.2.1 Introduction to the unified process

The unified process is used in this thesis as a guide for the creation of the RSS software. The sections that follow clarify certain concepts and features of the unified process necessary for its implementation.

According to Schach (2010), the unified process is a tried-and-tested iterative object-orientated methodology that is widely adopted in industry. One reason for its popularity is its adaptability, in that it may be tailored to suit different project types and sizes. The core concept of the unified process is the *iteration*, i.e., a self-contianed mini-project, characterised by a well defined result, that yields in a stable, integrated and tested release (Jacobson *et al.*, 1999). Further, each iteration, or mini-project, adds upon the previous release, incrementally building up to the final product.

The unified process, illustrated in Figure 4.4, possesses four life-cycle phases: inception, elaboration, construction, and transition, which place broad time segments on the life of a software project. Iterations are assigned across these phases during project planning, e.g., E_1 & E_2 denote two iterations within the elaboration phase. The distribution of effort and resources within each iteration can be seen by moving vertically down through an iteration crossing each of the unified process's workflows: requirements, analysis, design, implementation, and testing. At any point in time, an activity in the project can be classified as falling under one of the four phases, within a specific iteration, and engaged in one of the five workflows.

A brief discussion on the four key elements of the unified process is given below.

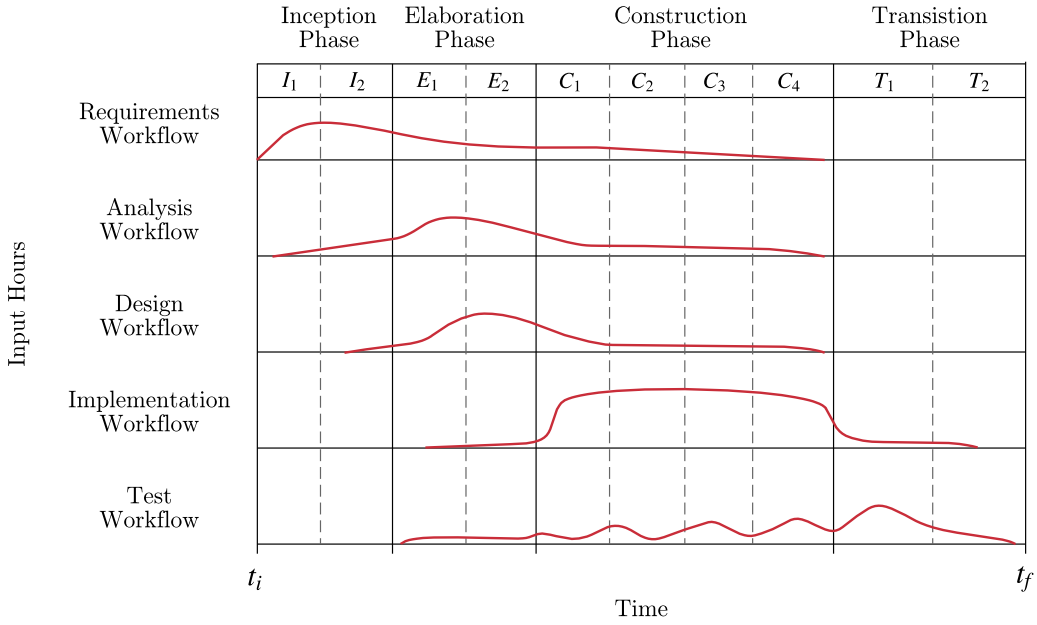


Figure 4.4: Phases of the unified process (Schach, 2010).

4.2.2 Four key elements of the unified process

Four elements underpin the unified process (Kruchten, 2003):

- Iterative and incremental: The unified process is based on the iterative and incremental life-cycle model (Section 4.1.2). The intrinsic advantage of this approach is that knowledge of one iteration is fed into all downstream iterations, mitigating the need for complete foresight into the project.
- Use case-driven: UML modelling is an enabler of the unified process. Use case models, specifically, serve as the primary input to the software creation process.
- Architecture centric: Emphasis is placed on thinking of the system not only in terms of it individual features, but also in terms of its global properties such as interoperability, modifiability, and scalability. Achieving such properties is only possible by intelligently considering the structures of the system early in the development process.
- Risk acknowledgement: As a feature brought forward from the Spiral model (Boehm, 1988), risk acknowledgement forces issues to be identified at the start of the project, in theory, reducing the number of problems later in the project.

4.2.3 Life-cycle phases

The life-cycle phases categorise a project's life into discrete time segments. Each of the four phases sets out guidelines to activities that should be performed, and specifies a milestone in the form of a project deliverable (Hunt, 2003).

1. Inception

   - In this phase the project is launched. The business case and initial requirements are defined and scoped, and iteration planning is performed. A good understanding of the problem domain is acquired and use cases that will drive the development process are outlined (Hunt, 2003). A prototype may be developed to demonstrate feasibility if required.
   - Deliverable: Project vision.
   - The RSS project was launched in **Chapter 2** where the first iteration of the inception phase was discussed (Section 2.4). Furthermore, an understanding of the problem domain was acquired during the construction of the simulation model (**Chapter 3**). The deliverables for the project vision are discussed in the chapters that follow.

2. Elaboration

   - User requirements are finalised and use case scenarios drawn up. Knowledge from domain and requirements analysis is used to begin the architecture. A series of analysis and design iterations are performed, leveraging a formal modelling process, to refine the structure of the system until a baseline architecture is achieved.
   - Deliverable: Baseline architecture.

3. Construction

   - The bulk of the development is performed in the construction phase. The end goal is a stable product, meeting all user requirements, and exhibiting the designed architecture. The product should be prepared for user acceptance testing. As the process is iterative, minor refinements to the architecture are still permissible.
   - Deliverable: Full beta release.

4. Transition

- The system is moved from the development environment into production. At this point it becomes available to the end users. A final verification of the clients requirements are performed, followed by project sign-off.
- Deliverable: Final release.

Iterations within each phase traverse through five workflows. The amount of time allocated to each workflow varies according to the project phase (see Figure 4.4) structuring the process, but also promoting the improvement and refinement of the system as multiple passes are made—the essence of iterative development.

4.2.4 Workflows

1. Requirements:
   - Aim is to determine the clients needs. This involves investigating the application domain, and understanding the setting in which the software is set to run.
   - High-level requirements were stated in Section 2.4. These requirements are broken down in the chapters that follow. Investigation into the application domain was completed with the assistance of the industry partner and the construction of a simulation of their gauge assembly process (**Chapter 3**).
2. Analysis
   - Analysis aims at resolving or decomposing the system in order to better understand it. This involves getting to know the problem domain, and capturing any business processes needed to support the requirements. Specific tools include: use case models, use case scenarios, and sequence diagrams.
   - The aim of this workflow is to refine the requirements until a comprehensive understanding of the underlying architecture is achieved.
3. Design
   - The design phase focuses on *how* the system should achieve its goal. Synthesising the analysis model to the point where the system is ready for development (Schach, 2010). This typically involves arriving at a formal internal structure of the system.
4. Implementation

- This workflow encompasses the coding portion of the project. Development teams are given control. The architectural design is implemented using one or more programming languages. The product is packaging and deployed on each iteration.

5. Test

   - Drawing up test cases and defining the testing activities so that the project can be continuously objectively assessed across its life.

In later chapters, the use of the unified process is demonstrated by its application. Furthermore, the role of the project phases, the allocation of iterations, and the progression through the workflows are discussed in more detail. Before this, attention must be given to the research intent of this thesis, specifying a software architecture.

4.3 Specifying the software architecture

An architecture is, essentially, a means to communicate software design decisions. More specifically, it depicts the structure of a software system, and defines the software elements that it is comprised of and the relationships among these elements (Bass *et al.*, 2012). Since architectures are generally complex, multiple views or abstractions of the system are required to partition or layer the complexity in an understandable form. In addition to depicting the structure and behaviour, a software architecture has consequences on system properties such as: robustness, modularity, comprehensibility, and performance (Kruchten, 2003). Thus, careful thought must be given to potential extensions or adjustments to the system in the future.

4.3.1 Software architecture description languages

The process of creating and documenting a software architecture, also know as an architecture description (AD), can be carried out in a number of ways:

1. Informal box and line notations.
2. Architecture description languages,
3. Unified modelling language (UML),
4. Object process modelling (OPM),
5. System modelling language, etc.

Informal methods allow the architecture of a system to be quickly and easily communicated between project teams, yet, their lack of formal notation allows for misinterpretation and are thus not considered viable for large software projects. Architecture description languages (ADLs) in contrast provide a formal specification for describing the structure and behaviour of the software. Notable examples include: Darwin (Magee *et al.*, 1995), Koala (van Ommering *et al.*, 2000), Rapide (Luckham *et al.*, 1995), and xArch/xADL (Dashofy *et al.*, 2001). Many of these ADLs were designed specifically for embedded systems engineering applications such as aerospace, automotive, and robotics systems, and have therefore not seen widespread adoption in enterprise software projects. In recent years, UML has become the de facto industry standard for specifying enterprise software architectures (Pandey, 2010).

In light of this, it was decided that UML would serve as the tool for creating the architecture description of the simulation-based RSS. Special attention was therefore given to ensuring that the software life-cycle and methodology complemented an architecture centric approach. As noted in Section 4.2.1, the unified process is underpinned by incrementally improving the architecture, and further more advocates the use of UML. To guide the process of using UML to beget an architecture, the UML modelling process of (Kendall & Kendall, 2011) was used.

4.3.2 UML modelling processs

An essential tool for the inception and elaboration phases of the unified process (and most other object-orientated methodologies) is the unified modelling language (UML) of OMG (2015). By prescribing a standardised syntax and semantics (specifiable objects), UML allows software systems to be effectively and efficiently specified and visualised. UML is itself not a software development process, but rather a modelling language used to support the software development effort. Further more, the many views and perspectives of UML standard allow it's outputs to be used as a common vocabulary for expressing object-orientated architectures.

Kendall & Kendall (2011) present a systematic approach to the use of UML diagrams in object-orientated systems analysis and design (OOAD). This approach, termed the UML modelling process, structures the way diagramming is done with the intention of making system complexity easier to navigate. Figure 4.5 shows the iterative procedure for UML modelling. Additional steps, including the creation of component and deployment diagrams, have been added to further support the unified process. The application of the UML modelling process if discussed in **Chapters 5**, **6**, and **7**

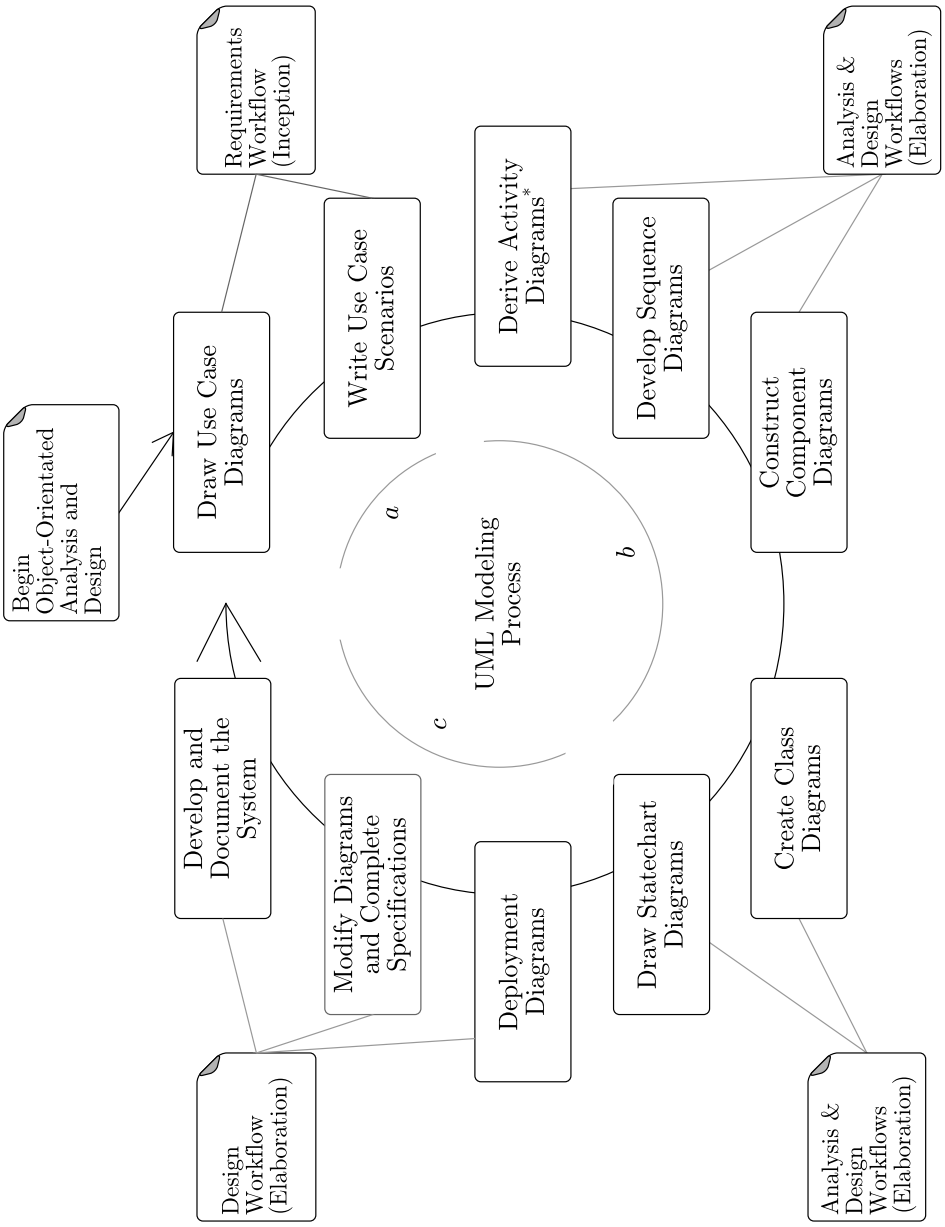


Figure 4.5: Iterative cycle of the UML modelling process (Kendall & Kendall, 2011).

Since OOAD is integral to UML modelling some advantage are mentioned here. In contrast to structured analysis where operations and data are considered separately, OOAD actively groups, or encapsulates, data and operations (methods that act on data) into objects. The primary advantage being object reuse, thus making the system more modular (Schach, 2010). Another advantage is object replacement—granted that the object's interface remains unchanged. This is particularly important in the Web and mobile spaces because of the rate of evolution of the supporting technologies. OOAD therefore, by its fundamental principles, makes provision for the ever changing Web and mobile landscapes; and in general assists with the goal being to create a reliable, extensible, and evolvable software systems. Furthermore, the iterative process of analysing and designing a software system using UML modelling process simultaneously documents the architecture of the system.

4.4 Concluding remarks on Chapter 4

This chapter began by introducing and discussing three popular software development life-cycle models. After evaluating the advantages and disadvantages of each, the iterative and incremental model was selected as the model of choice for the construction of the software components of the reactive scheduling system in this thesis. The unified process, a specific implementation of the iterative and incremental model, was then discussed at length; it four key elements were introduced, the project phases were defined, and the five workflows were described. Finally, the UML modelling process (Kendall & Kendall, 2011) was presented and its property of documenting software architectures was highlighted.

Chapter 5

Tracking WIP using a sensor network

Ensuring the simulation model remains synchronised with the structural configuration and dynamic state of the real system is an important aspect of a simulation-based reactive scheduling system (RSS). The manner in which the simulation model is kept up-to-date with the happenings of the real system is dependent on a variety of factors, including: the characteristics and constraints of the manufacturing environment (e.g. rescheduling environment), the features or limitations of the simulation software, and the architectural choices made during the design of the scheduling system.

To fully capture the system state at any one point in time, numerous internal and external disturbances of the manufacturing system would need to be monitored in real-time. For example: operator positions, machine statuses, inventory levels, job backlogs, work-in-process (WIP) locations, new orders, etc. would all need to be captured and stored as a system 'snapshot' (or, a digitally encoded state of the real-system). However, acquiring a fully qualified snapshot the system is often unnecessary as certain event classes may not be relevant to the goal of the system, and many are likely to have a low contribution to the objective of the system.

The selection of dispatching rules as the rescheduling method (Section 3.2.1) imposed a requirement on the data needed in the system snapshot. Since dispatching rules operate by dynamically ranking tasks in workstation input buffers, the contents of the input buffers had to be know. In other words, the system snapshots would need to include data points on the location and state of WIP items. The challenge of designing and developing a WIP monitoring solution is the topic of this chapter.

5.1 Designing a WIP tracking solution

The construction of a WIP tracking solution is intended to meet the second requirement of a data acquisition platform discussed in Section 2.4.3. The requirement stated that internal manufacturing disturbances should be captured at a sufficient level of detail to afford the simulation model the ability to initialise off of a state snapshot. To determine the level of detail, or resolution, of WIP events needed for the snapshot, an investigation into the movement of WIP at the industry partner was conducted.

5.1.1 Examining WIP state transitions

As described in Section 3.3.1, the industry partner's operation focused on the assembly of pressure gauges. Each batch of gauges (referred to as a task) was placed on a polystyrene pallet to assist with inter-workstation transfers. Operators were obliged to complete a task before beginning the next task, i.e. all gauges on the pallet needed to be assembled before the next task could begin, or before the task could progress to the next workstation. Considering this, it was evident that tracking WIP items should happen, not at the level of the gauge, but at the level of the task.

In addition to inter-workstation transfers, tasks underwent a series of state transitions at a each workstation. A typical state transition sequence was as follows (see Figure 5.1). A task arrives at a workstation (t_0), and enters the input queue awaiting operator intervention. At time t_1 the workstation's operator selects the task and begins the setup activity. At time t_2 processing of the task begins, at time t_3 processing is completed and the teardown activity begins, completion occurs at time t_4, and dispatch—the final state transition—occurs at time t_5, after which the task is en route to the next workstation.

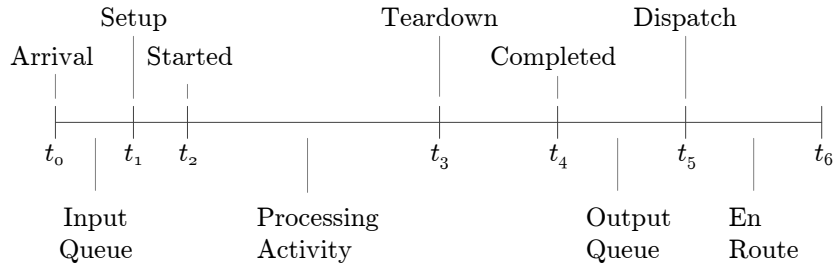


Figure 5.1: Workstation WIP state transitions.

As mentioned, the use of dispatching rules demanded the contents of workstation input buffers be known to perform task ranking. Thus, the *arrival* and *started*[1] events of participating workstations had to, at minimum, be captured.

With the addition of each state transition, the event resolution of the workstation increases, i.e., a finer level of detail on the workstation is gained. The number of states to be captured is influenced by three factors: a) the desired level visibility into the system, b) the WIP mapping requirements (discussed in Section 5.1.2 below), and c) for manual capture systems, the operator-node interaction trade-off (discussed in Section C.1.1).

5.1.2 Mapping WIP to the simulation model

The primary[2] purpose of tracking WIP was to provide an up-to-date snapshot of the shop floor for the simulation model to *initialise* from. Achieving this involved constructing a mapping between the WIP items on shop floor and those in the simulation model. In other words, the event resolution of the WIP tracking solution had to match, or at least map to, the event resolution of the simulation model.

For example, the capturing of the 'arrival' event of of task, T_j, at workstation, W_m, on the shop floor means the simulation model be capable of placing a virtual task, T_j, in the input buffer of virtual workstation,W_m, upon initialisation. This proved challenging as the device capturing the WIP transition, the the simulation model, and the systems in between (Web service, state database, RSA) all needed to needed to be designed simultaneously. As this example demonstrated, the application of the industrial engineering mantra of 'systems thinking' was crucial in arriving at a feasible solution.

5.1.3 Requirements of a WIP tracking prototype

For the purposes of this thesis it was decided that four state transitions: 'received', 'started', 'completed', and 'dispatched' would be captured to sufficiently illustrate the state capture concept. With regards to the simulation model's WIP mapping implementation, WIP items in state 'received' and 'started' were positioned in the input buffer of the respective virtual workstations, and items in state 'completed' and 'dispatched'

[1] Although sequence-dependent setup and teardown activities were common, these activities were ignored as their contribution to the processing activity at each workstation was found to be less than one percent of total processing time of a task.

[2] The secondary purpose was to provide visibility into the system, e.g., location of WIP items (see Section 2.4.3).

were placed in the output buffer of their respective virtual workstations. The result is that the simulation model was initialised with a good approximation of the state of the shop floor at every rescheduling point.

To achieve the state capture phase, a set of technologies needed to be found which had the capability to:

1. *uniquely identify* both WIP items and workstations, and capture the:
2. *transaction* between WIP items and workstations, including the
3. four *state transitions* of the WIP item at the workstation, the
4. *timestamp* of the state transition instant, and finally
5. *communicate* these data to the RSA.

In doing so, each captured transaction would contain enough information to identify the task, its current location, state, and the timestamp of the instant it transitioned into that state.

5.2 Prototyping the sensor network

Before a WIP tracking solution was developed, an analysis of the as-is state of the industry partner's infrastructure was conducted. The findings were as follows:

1. no manufacturing execution system was installed,
2. no industrial communication backbone (e.g. RS-232, MODBUS, CANBUS, etc.) existed, and
3. no autonomous or computer-controller equipment was used.

The fact that no existing information infrastructure could be integrated into, or at least used to facilitate data transfer, meant the prototype needed to operate independently of the assembly environment.

A review of literature revealed many solutions to WIP tracking, such as, mechanical detection systems, advanced machine vision systems, Bluetooth beacons, etc. Radio frequency identification (RFID) was found to be a popular and proven means of tracking WIP with many successful real-world implementations (Chongwatpol & Sharda, 2013).

Attributes such as: durability, and re-usability, and low cost, has let to RFIDs widespread use in industrial applications. for further information the reader is directed to Guo *et al.* (2014) and Groover (2008) .

Three goals governed the conception of the prototype: RFID-based, independent communication infrastructure, operator assisted transaction captures (to shorten development time).

The hardware and software components of the node are presented separately in this chapter for readability; but it should be noted the prototyping phases of analysis, design, development, and testing occurred simultaneously. The iterative process of the unified process was applied throughout the prototypes life-cycle—described in more detail in Section 5.3. The following sections described the hardware components of the nodes.

5.2.1 Hardware components of the node

With the goals of the prototype in mind, a solution was arrived at involving a wireless sensor network (WSN). This WSN was comprised of three nodes, where each node was capable of operating independently of its peers. The nodes allowed a transaction to be captured by workstation operators (the capture was semi-automated). Communication of the transaction to the reactive scheduling application (RSA) occurred via central a central router (which itself forwarded packets to a Web service (discussed in Section Cross Ref: Web service)). Since the WSN communicated using self-generate electromagnetic waves, no physical communication infrastructure was needed.

5.2.1.1 Microcontrolller

The core component of the node was a Waspmote micro-controller. With two UART ports, and numerous off-the-shelf modules, including RFID, the Waspmote served as the basis off of which the node was built. The dual UART interfaces allowed two communication modules to be connected simultaneously—an important feature for this prototype—and a contributing factor to its selection over other popular prototyping platforms, e.g. Arduino, Beaglebone, etc.

The purpose of the Waspmote was to:

- Serve as the central processing unit,
- Interface with RFID and WiFi modules,
- Uniquely identify a workstation by way of its internal serial number.

Attached to the first UART socket on the Waspmote was a 13.56 MHz RFID module.

5.2.1.2 RFID module

The RFID module allowed for the identification of RFID cards, using radio frequency, as they entered a 'read-zone' between zero and five centimetres above the antenna. The purpose of the module was to identify an RFID card when the read-zone was activated. Four buttons, each of which corresponded to the modelled state transitions of 'received', 'started', 'completed', 'dispatched', activated this read-zone upon a 'touch-down' event. That is, when one of the state buttons was pressed, the RFID module activated and searched for an RFID card.

RIFD cards were used to uniquely identify WIP items (tasks). Interaction between the operator and the node, and the RIFD module and card, resulted in four pieces of information being collected: Waspmote Id, card Id, state transition, and timestamp timestamp. This combination allowed the RSA to know the location of a WIP item at that point in time. This choice of a semi-automated identification and data capture mechanism allowed for the digitisation of state information, without the need for manual entry into a computer.

5.2.1.3 WiFi module

Attached to the second UART of the Waspmote was a 2.4 GHz b/g/n WiFi module. This module facilitated the communication between the node and the Web service through HTTP requests (via a central router).

The purpose of the WiFi module was to:

- Establish a connection to the primary node (central router).
- Initiate HTTP Web requests.
- Communicate captured state transition events the Web service.

Subsequent research revealed IEEE 802.11 b/g/n radios may be subject to interference in certain environments from electromagnetic waves produced by industrial equipment. The use of IEEE 802.15.4 (Zigbee), which has displayed less interference in such environments and realises a lower power draw (Baker, 2005; Lee *et al.*, 2007), is now recommended for most industrial applications. The use of IEEE 802.15.4 is therefore recommended for future work.

5.2.2 Procedure for capturing a WIP transaction using the node

To illustrate the procedure for capturing a WIP transaction, consider the workstation (typical of the industry partner) shown in Figure 5.2. Six tasks currently reside at the workstation. Each task is uniquely identified by an RFID card to which it has been paired. The workstation itself is uniquely identified by the node placed at the workstation.

When a state transition occurs, for example, task 4 002 moves from the input buffer to the processing station, the designed capture procedure for the operator is:

1. *Press* the button corresponding to the state transition, i.e. 'started' (activating the RFID read-zone).
2. *Swipe* the RFID card, paired to the task, over the antenna (identifying the RIFD card).

This two step process is all that is required from the operator to capture the transaction.

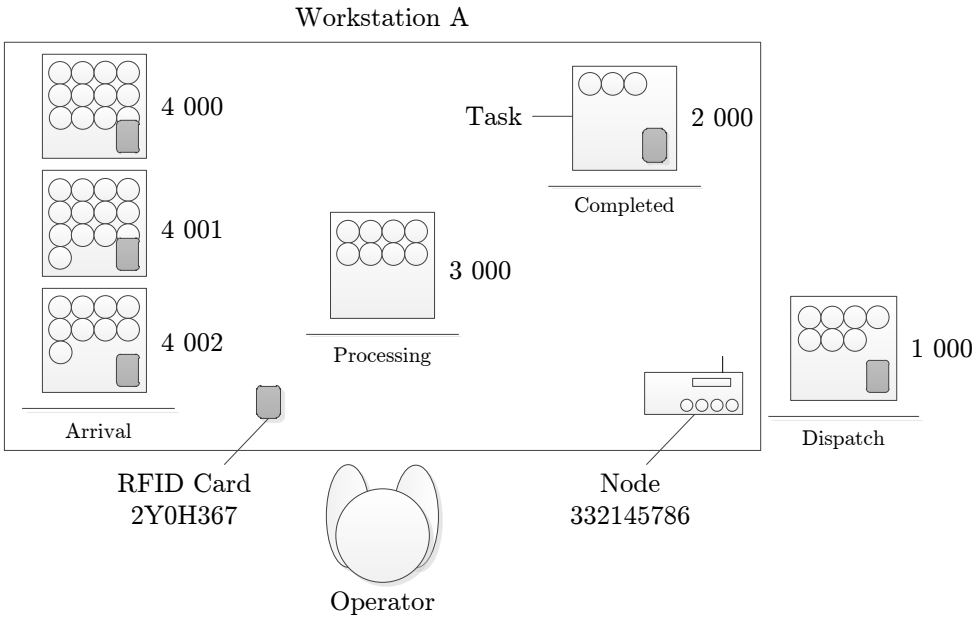


Figure 5.2: Workstation showing a typical loading.

Communication of the event to the Web service was programmed to occur off the back of the capture event (discussed in Section 5.3) and requires no operator interaction.

5.2.3 Summary of the node prototype

Table 5.1 contains references to a number of properties and aspects of the node prototype (after completion of the final iteration). It includes the technical specifications of the components described above, as well as a bill of materials. Additionally, a photograph and a design mockup are shown to give the reader an impression of the prototype and the vision which inspired it.

Table 5.1: Aspects of the node prototype.

Item	Reference
Waspmote technical specifications	Table C.1
RFID module technical specifications	Table C.2
WiFi module technical specifications	Table C.3
Bill of materials	Table C.4
Photograph of the prototype	Figure C.3
Concept mockup	Figure C.4

The mockup design was created as part of the 'project vision' deliverable of the inception phase of the unified process (see Section 4.2.3).

5.2.4 Managing entity relationships

To support and maintain the dynamic relationships between entities in the real system and the software representations of those entities, e.g. workstations, nodes, WIP items, RFID cards, etc., an information system was designed and built. This information system is presented in **Chapter 7**, but points important for this chapter are given below.

Table 5.2 summarises the component associations maintained in the RSS state database. An EERD of the RSA database showing the entities and their relationships is shown in Figure E.17.

Table 5.2: Component associations

RSS component		Real system
Node (Waspmote)	$\longleftrightarrow$	Workstation
RFID Card	$\longleftrightarrow$	WIP item (task)
Button	$\longleftrightarrow$	WIP transition state

The system was designed so that nodes and cards could be dynamically associated with a workstation or a WIP item. In other words, nodes and card were placed in pools from which they could be drawn and returned. Further, each task was automatically unassigned from its associated card when the completion event of the final operation in its assembly sequence occurred.

5.3 Developing the node software

In the first iteration of the requirements workflow (project inception), general requirements for the data acquisition platform were formulated. These requirements were further refined to suit the industry partners' needs during a second iteration discussed in Section 5.1.3. This section deals with the construction of the node software from these requirements.

5.3.1 State capture, synchronisation, and initialisation

The process of transferring the current state of the shop floor to the simulation model occurred in three stages as shown in Figure 5.3, and described below:

1. **State capture**: Data points from WIP transactions were captured by workstation operators using nodes of the wireless sensor network.
2. **Syncronisation**: These data points were pushed to the RSA by consuming the WIP endpoint of the Web service. After parsing the data, the RSA simultaneously notified the event filter of the event and persisted the transaction data to the state database. The RSA was responsible for updating the data snapshot in the state database ensuring an up-to-date picture of the assembly operation was continually available.
3. **Initialisation**: Incoming white-listed events (following the rescheduling policy, see Section 2.5) went on to invoke the simulation optimisation engine. At simulation model compile time, the state data was extracted from the state database and piped into a set of relation tables built into the simulation model. The model initialised itself from these relational tables before executing.

Development of the node software covers the data capture phase (i.e., supporting the state capture procedure) and deals with the forward leg of the state synchronisation phase.

The state initialisation phase is handled in more detail in **Chapter 7** during the description of the simulation optimisation engine.

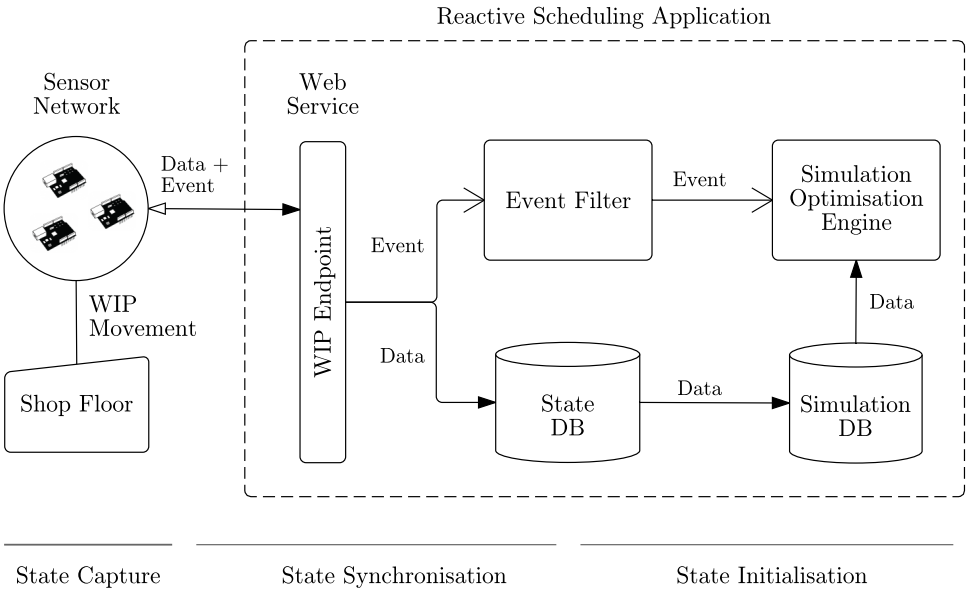


Figure 5.3: State capture, synchronisation, and initialisation.

5.3.2 Application of the UML modelling process

The unified process advocates software creation that is driven by use case models (Section 4.2.2). Use case models, such as the one shown in Figure 5.4, specify *what* the system should do, specifically with respect to its external interfaces; and are intended to illustrate high-level system functionality (Hunt, 2003). The construction of the use case model formed part of the second[1] iteration of the requirements workflow.
Figure 5.4 illustrates the use cases for the node prototype. These cases map back to the textual description of the procedure for capturing a WIP transition (Section 5.2.2). The system boundary demarcates the scope of functionality included in the node prototype. Actors in the system were identified as workstation operators as they are solely responsible for operation of the node. The primary use case is the capturing of a state transition, which includes identification of the WIP item and the state transition it has just experienced.

After completion of the use case diagram, a use case *scenario* was created. This scenario, shown in Table C.5, was used to further the refine the procedure for capturing a WIP transition. The order of the interaction between the operator and the node, the

[1]The first iterations was completed during project inception, see Section 2.4.2

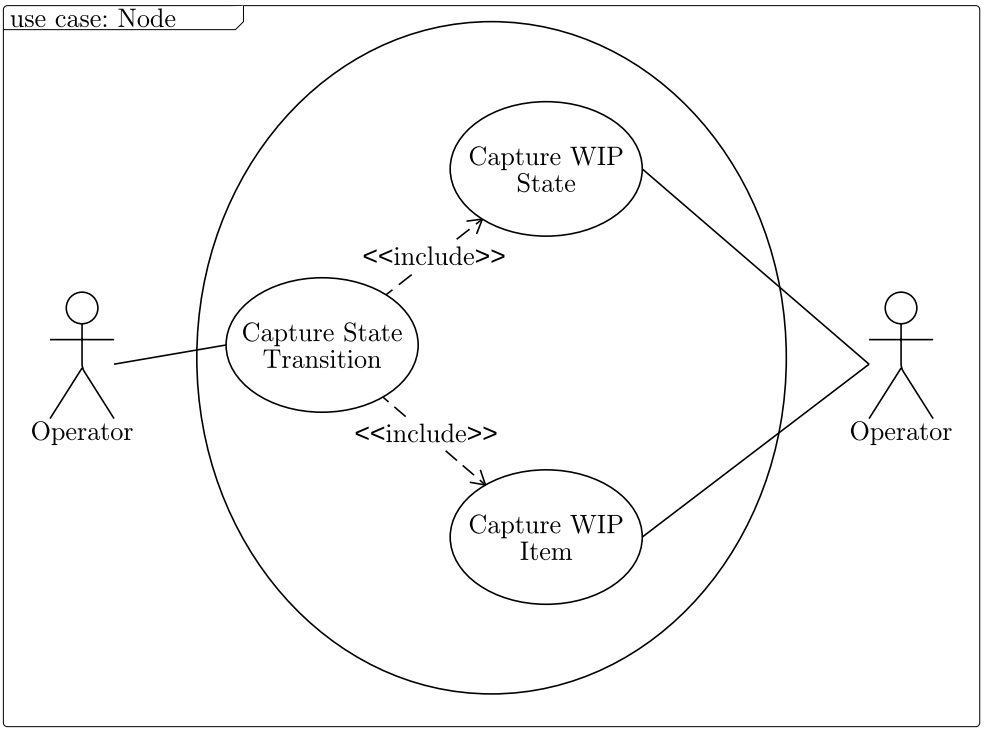


Figure 5.4: Use case model of the node.

data fields needed, and other properties, such as pre and post conditions of the node were logged.

5.3.3 Analysis and design workflows

At this point the procedure of interacting with the node was understood the associated software processes that would need to occur highlighted. This concluded the node inception phase. The elaboration phase focused first on modelling the domain problem and secondly on designing system. The relevant UML diagrams are discussed below.

- Data flow diagrams[1]: After defining the use case scenario, the processes of the node were analysed by decomposing the system from the top down. This can be seen in the Context and Level 0 data flow diagrams (DFD), shown in Figures C.5 and C.6. The final, and most detailed DFD, i.e. Level 1, is shown in Figure 5.5. In this diagram, the entities: RFID card, Node interface, and Web service, are shown interacting with the processes of the system, e.g. retrieve card serial id. The data flows between these processes are also shown.

- Sequence diagrams: To illustrate the order of events, and the messages exchanged between the set of participating objects, sequence diagrams were created. The first iteration is shown in Figure C.7 and the second and final iteration in Figure C.8. Where the DFDs revealed on the primary processes, sequence diagrams revealed the order in which data would need to flow between entities.

  The UML diagrams discussed thus far have described the behavioural aspects of the objects of the node, i.e. their dynamic behaviour over time. Attention is now given to the static structure of the node, i.e. object relations and physical deployment.

- Component diagrams: The physical structure of the code base was arrived upon by constructing a component diagram, shown in Figure C.9. As can be seen in the diagram, the node software leverages two third-party libraries: `WaspRFID.h` and `WaspWIFI.h`. These libraries provided interfaces which abstracted away much of the low-level programming necessary for interacting with the RFID and WIFI modules over UART.

[1] Data flow diagrams (a structured analysis and design technique) were constructed as a replacement for UML activity diagrams as, in the authors opinion, DFDs provide greater insight into the model domain—whereas activity diagrams focus on control flow (an aspect already covered by the UML sequence diagram).

Care was taken to ensure the developed components communicated with the software libraries in a stateless manner, i.e. no hard-coded variables or references were used. This allowed the same code base to be deployed to each node without the need for reconfiguration—an important property of scalability.

- Deployment diagram: The physical architecture of the prototype, displaying the hardware components discussed above and their communication protocols, is shown in a UML deployment diagram, see Figure C.10.

With node prototype hardware finalised, and iterations across the analysis and design workflows of the elaboration phase completed, construction of the node software could begin.

5.3.4 Construction of the node software

Construction of the node software occurred over three planned iterations. The first two iterations focused on the transaction handles component (see Figure C.9). Whereas the final iteration focused on the Web service manager component.

1. Base functionality:
   - Accessing the Waspmote's internal serial identifier.
   - Creating an edge detection class to notify the node when and which state transition button was pressed.
2. RFID communication:
   - Establish communication with the RFID module.
   - Access RFID card unique identifier using the RFID module.
3. WIFI communciation:
   - Establish an HTTP connection.
   - Consume the RSA Web service.

To communicate to the REST Web service over HTTP, a URL query string was constructed in the following format:

"`/api/wip/event/?a=CardId&b=NodeId&c=State`".

where the path parameters: *cardId*, *nodeId*, and *state* contained the captured transaction parameters[1]. The endpoint definition can be found in Section F.3.

[1]The capturing of timestamp was handled by the server (further discussed in Section C.1.2)

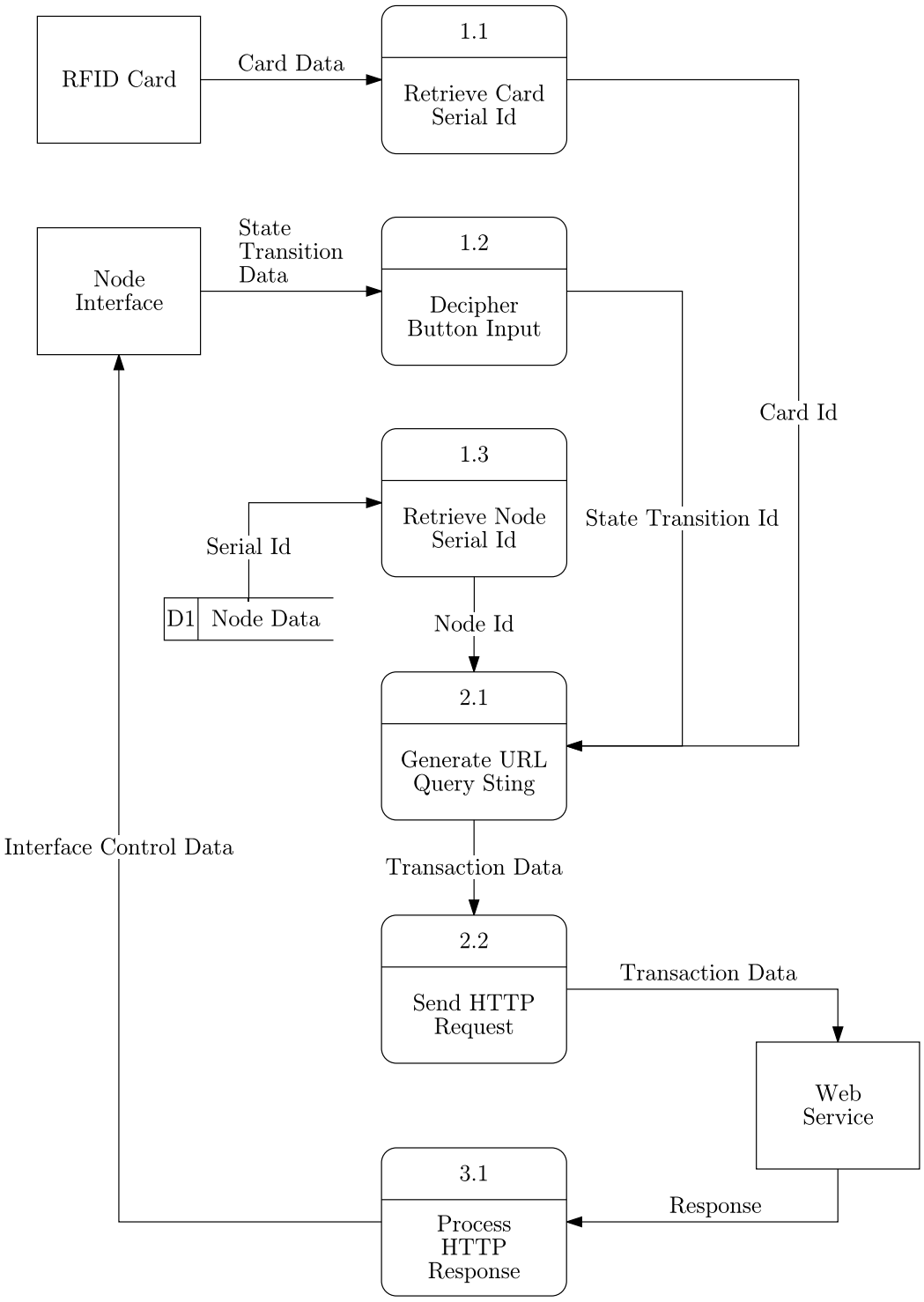


Figure 5.5: Data flow diagram: Node — Level 1.

5.4 Verification and validation

5.4.1 Face validation

Face validation was used to ensure the node was responding appropriately to the state capture procedure. This was done through visual inspection.

Table 5.3: Software test cases.

Function	Criteria	Validated
Neutral state	Node remains in *neutral* state while powered on.	√
State buttons	Node enters *capture* state when any state transition button is pressed.	√
RFID card	LED blink pattern seen for successful RFID card unique Id acquisition.	√
	LED blink pattern seen for successful RFID card unique Id acquisition.	√
Networking	LED blink pattern seen on server response.	√

5.4.2 Software test cases

A set of test cases to check the output and functioning of the node software was written. This was done by inserting `printf` statements at certain points in the code logging variables of interest to the serial monitor of the Waspmote IDE. A summary of these cases is shown in Table 5.4.

Table 5.4: Node face validation criteria.

Function	Criteria	Validated
State buttons	Edge detection function registers voltage change on touch done event.	√
RFID module	RFID module initialises correctly and without error.	√
	Unique identifier of RFID is captured when passing card over the read-zone.	√
WIFI module	WIFI module initialises correctly and without error.	√
Networking	URL string is formulated correctly with correct path parameters.	√
	Server response of HTTP 200 (OK) is received on success.	√

Continued on next page

Function	Criteria	Validated

5.5 Concluding remarks on Chapter 5

In this chapter a wireless sensor network was presented as a solution to challenge of tracking WIP movement. This addressed data acquisition requirement of capturing the internal disturbances of the manufacturing system at a sufficient level of detail to enable simulation model state initialisation.

To this end, a node prototype was presented as a potential means of capturing WIP transitions at workstations. The hardware components of the node were presented and the procedure for the capturing of WIP events was examined. Furthermore, the node construction of the software was discussed, and the application of the unified process was demonstrated. The UML was used to capture design architectural decision decisions.

Chapter 6

System management and control using a mobile client

6.1 Requirements for the mobile client

After observing the daily operations of the industry partner it was evident that two further system disturbances (in addition to the work-in-process (WIP) movement captured by the sensor network of **Chapter 5**) needed to be collected to further improve the system snapshot. First, the list of jobs, set for imminent release onto the shop floor, would need to added as they would naturally affect the assembly operation once released. Secondly, expedited orders were common to the industry partner[1], and therefore needed to be modelled. When an order was expedited, operators treated the order as having the highest priority, and selected it over any other tasks awaiting processing. By capturing these these two external disturbances (and monitoring of WIP movement) a system snapshot adequate for simulation model initialisation could be created—thus, satisfying the reactive scheduling system's (RSS) requirement for the data acquisition platform (Section 2.4.3).

6.1.1 Decision support platform

The RSS requirement of creating a decision support platform (DSP) (Section 2.4.6) was also met using the mobile client. As discussed, the primary goal of the DSP was to communicate the selected dispatching rule to the shop floor for execution[2]. In doing

[1] Many of the industry partner's clients operated pressure sensitive equipment and were eager to maintain good levels of gauge safety stock.

[2] See Section C.1.3 for a discussion on the use of the mobile client to display the task recommendation.

so, the operation of the shop floor could be controlled and its output governed. The secondary, and optional, goal of the DSP was to provide managers with meta-data about the system. The purpose this data being to assist in decision making by giving managers better visibility into the system.

Since the aim of this thesis was to design and develop a *practical* reactive scheduling system, the industry partner was consulted to obtain realistic meta-data requirements, in doing so ensuring the meta-data selected were relevant to industry.

6.1.2 Documenting the requirements

As prescribed by the UML modelling process, requirements were gathered by building out a use case model to describe *what* the mobile client should do. This use case, shown in Figure 6.1, depicts the actors (floor managers and operators), and the high-level events (mapping to the requirements for the data acquisition platform and the DSP)[1].

Use cases the industry partner was particularly interested in were: 1) viewing the location of jobs in the system at any point in time, and 2) being able to determine each job's estimated completion time (a property the industry partner found difficult to estimate). In addition to these meta-data requirements, a further use case was added: 'Trigger SOE' (shown in Figure 6.1). This use case modelled the rare event in which the floor managers would need to manually override the SOE and force the simulation experiment to reinitialise and execute. Performing this action would cause all operation input buffers to be refreshed according to the dispatching rule selected by the SOE.

After completion of the use case model, use case *scenarios* were developed to further describe the actors' interaction with the system. Scenarios for 'Add job to system', 'View estimated job completion time', and 'Trigger SOE' are shown in Tables D.2, D.3, and D.4 receptively. Only the primary flows are shown, i.e. the standard or typical flow of events through the system.

[1] A structured view of the requirements is also shown in Table D.1.

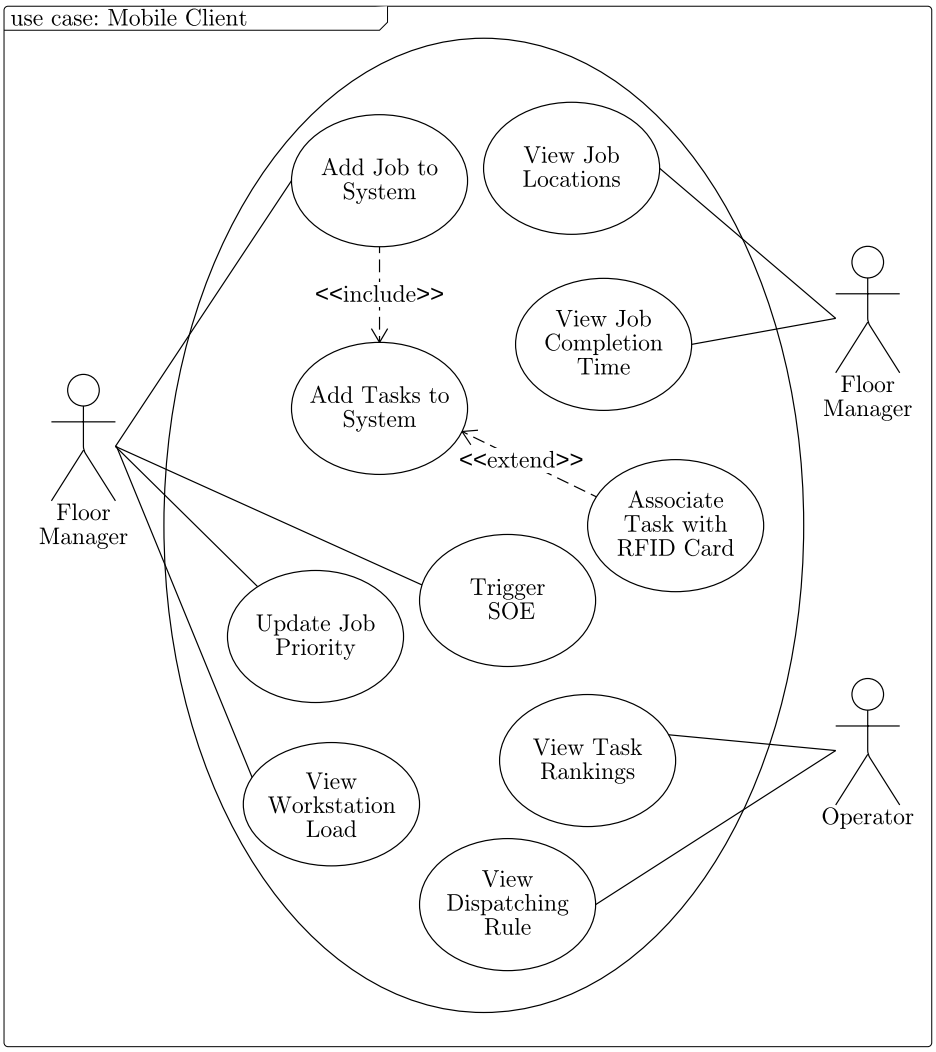


Figure 6.1: Use case diagram: Mobile client.

6.2 Mobile application architecture

Due to the complexity of creating a mobile application, a portion of the inception phase was dedicated to research into the mobile development domain, specifically, application architectural design patterns. The outcome further structured the UML modelling process.

6.2.1 Model-view-controller

The architectural pattern of model-view-controller (MVC) was found to be fundamental to all major mobile platforms. This architecture, simplified in Figure 6.2, proposes a software design pattern where application objects are separated into three categories, each with a specific role, and each of which communicate with objects in the other categories in a defined and structured manner. The three object categories are (Fowler, 2003):

- Model: Model objects encapsulate application data and define any computation that is set to manipulate the data. Data that must be persisted are stored in model objects. A model may never interact with a view directly.
- View: Any objects whose data users can see are classed as view objects. This includes the set of user interface controls and components, e.g. buttons, labels, icons, etc. A view may only work with data provided to it by a controller.
- Controller: Controllers relay communication between view objects and model objects. This includes: managing the life-cycle of on screen objects, communicating with the Web, executing business logic, etc. Controllers *update* views and user actions *notify* controllers.

MVC reinforces the idea of a separation of concerns, and therefore imparts simplicity on the system by its application. Further benefits include keeping the source code clean, promoting modularity by decoupling the user interface from the representation of the data, and improving the testability of the application.

6.2.2 Delegation

The principle of delegation is reliant on the concept of a protocol. Protocols declare programmatic interfaces (consisting of methods and properties) that a class may select to subscribe to. Once an object has subscribed, that object acts on behalf of the delegating object. Delegation is a means of enabling the MVC pattern as it allows objects, such

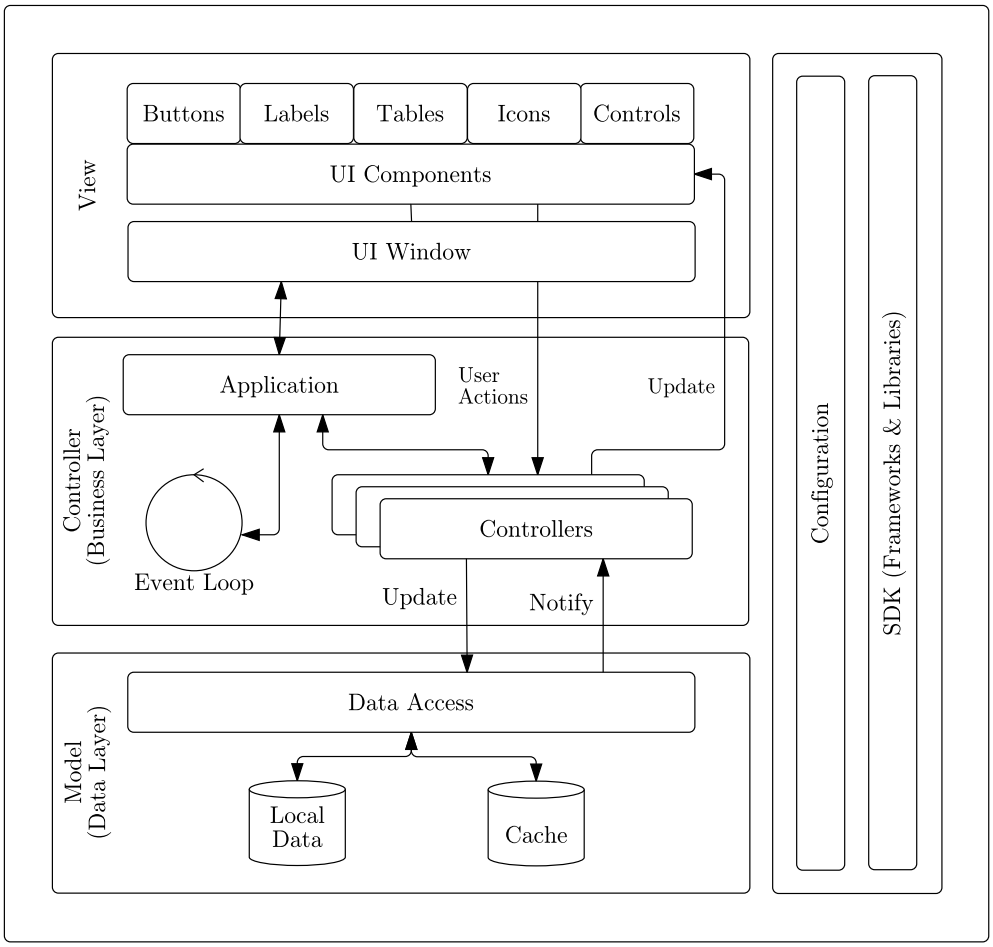


Figure 6.2: Model-view-controller architectural design pattern.

as tables, to delegate computational tasks, e.g., populating their cell contents, to a controller. This concept is demonstrated in Section 6.4.1.

6.3 Designing the mobile application

After the investigation into mobile application architectures and design patterns was completed, the design of mobile application began, starting with the selection of a device and platform on which to develop.

6.3.1 Device and platform selection

When selecting the device on which to prototype the mobile client, it was determined that two minimum features had to be met.

1. Connect to the internet over using a standard protocol, e.g. TCP/IP, HTTP, etc.,
2. Possess a screen size equal to or larger than 8.89 cm (3.5 in) (see Section D.1.2).

The selection of a mobile application platform was not of marked importance since most mainstream mobile operating systems (e.g. Android, iOS, Windows Mobile, etc.) included fully-featured stable software development kits (SDK). Furthermore, each of these SDKs enabled control of their respective devices through sufficiently abstracted APIs, e.g. application life-cycle management, networked communication, on-device data persistence, and display control.

6.3.2 Analysis and design workflows

The analysis and design workflows of the elaboration phase were conducted over two iterations. References to relevant UML diagrams, and descriptions of the important points are given below.

- Sequence diagrams: In the first iteration of the workflow, the order of interaction between the user and the mobile application, and the mobile application and the RSA Web service was determined, as shown in Figure D.2. During the second iteration the interactions and messages were further refined, see Figure D.3 (RESTful HTTP verbs are also shown).
- Timing diagrams: To further explore the interaction between the entities, and more specifically to understand the entity *states* at run-time, timing diagrams were created. An example in shown in Figure D.5.
- Component diagrams: The physical structure of mobile application is shown in Figure D.6. A component worth highlighting is the native SDK object, 'application delegate', this helper object performs many application-wide tasks and is

responsible for notifying other object of important application life-cycle events, e.g., application will start-up, application will shut-down, etc.

- Deployment diagram: The compiled iOS source code, know as an application bundle, was deployed to a third generation Apple iPad®. Deployment of the application bundle within the context of the RSS is shown in Figure E.14 (Appendix E).

6.3.3 User interface mock-ups

A functional view was taken when designing the user interface of the application, i.e., only those elements needed to meet the functional requirements were included. For a real-world system, it is recommended that modern user interface approache be followed, e.g. user centred design. Table 6.1 provides references to user interface mock-ups created for the mobile client.

Table 6.1: References to use case interface mock-ups.

Use case	Reference
View job locations	Figure D.7
View job estimated completion time	Figure D.7
Add a job	Figure D.8
Adjust job priority	Figure D.9
View workstation load	Figure D.10

The specification of the requirements and completion of the interface mock-ups concluded the elaboration phase. This was followed by the construction of the mobile application.

6.4 Developing the mobile application

The development device chosen for the mobile client was a third generation Apple iPad®. The device's from factor, with a display of 24.63 cm (diagonal), offered adequate screen real estate for data input and the display of decision support data. Further, the native iOS operating system, and proven application platform were desirable traits. Those aspects that were important during development are listed below:

1. The adoption of the MVC architectural pattern was paramount.

2. UML diagrams created during elaboration assisted in logically structuring the code-base.

3. Special attention was given to threads and concurrency. Lengthy operations, such as network access, needed to be offloaded to subordinate threads to prevent the user interface (processed by the main thread) from becoming unresponsive.

6.4.1 Application of the MVC design pattern

The sequence diagram of Figure D.4 demonstrates the application of the MVC architectural pattern (Section 6.2.1). The figure shows the communication between application controller and the Web service, and the intra-communication between three application objects: job table view (left-hand table of Figure D.7), job controller, and the job model. The controller was set as the delegate of the table view, and to interact with the model to retrieve data. When the user actioned to view the table, the controller was messaged to retrieve the necessary data, number of rows, cell contents, etc, and return it to the table. Thus, the controller acted as the delegate of the table view.

6.4.2 Integrating with the RSA Web service

Both the sensor network nodes and mobile client (front-end devices) interacted with the RSA via a Web service.[1]. This service exposed endpoints over the internet which served and the entry points to the data and capabilities of the RSA. Table 6.2 gives a list of the endpoints that were exposed off the RSA.

Table 6.2: RSA Web service endpoints.

Resource	Description	API Endpoint
Job	Job creation. Priority update. Fetch all Jobs.	Section F.1
Task	Fetch all tasks and queue positions.	Section F.2
Operation	Fetch all operations and associated load.	Section F.4
RFID	Fetch all available RFID cards.	Section F.5
SOE	Manual invocation of the SOE.	Section F.6

In terms of a data exchange format, RESTful Web services predominantly use the JavaScript object notation (JSON), defined by RFC 4627 (Crockford, 2006), for the body of HTTP messages. This arrangement is commonly referred to as JSON over REST).

[1]Details on the design of this Web service are provided in Section 7.3.2

JSON provides a simple language-independent way of formatting program specific data structures (arrays, dictionaries, etc.) as simple strings. Clients can then consumes the Web service by send and receiving JSON formatted strings.

These strings may then able to be sent over the internet to a client where the data can be parsed into native data structures. This format has displaced the other formats, e.g. XML, as it is both a lightweight syntax and human readable with the need for parsing.

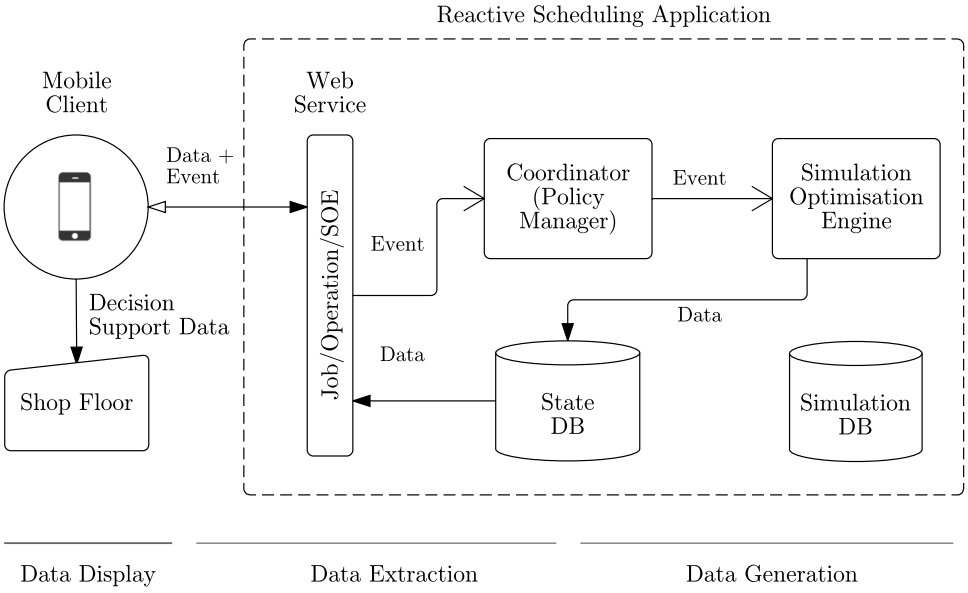


Figure 6.3: Conceptual architecture — Decision support.

By way of an example, see Figure 6.3, when the floor managed updated the priority of Job, the client formulated an HTTP request with a JSON representation of the job resource including the updated priority (Section F.1). This request was sent to the Web service which triggered the relevant internal processes in the RSA. In the case where the event was whitelisted, the SOE would be triggered and new control instructions generated. If all processes executed successfully, the new representation of the job resource was returned[1]. to the client in JSON format where it was deserialised

[1]Responses returned by the server were programmed in accordance with the RFC 2616 HTTP/1.1 specification (Fielding *et al.*, 1999), i.e. responses included HTTP status codes and the location of the resource via its URI.

and pushes to the user interface. A second request to fetch update decision support data was issued immediately afterwards.

6.5 Verification and validation

Face validation was used to ensure the mobile client responding appropriately to user interaction with the app. This was done by manually by interacting with the application through visual inspection.

Table 6.3: Mobile client validation criteria.

Function	Criteria	Validated
App launch	App launches and presents home screen.	√
Job table	Job table shows jobs in states 'NEW', 'WIP', and 'COMPLETED'.	√
	Job expected completion times are shown in the correct job row.	√
	When a job is tapped, its corresponding sub tasks are shown.	√
	Adding a new job leads to a data refresh and the new job is shown in the table.	√
Task table	Task table shows tasks in states 'NEW', 'WIP', and 'COMPLETED'.	√
	The current location of a task is shown (if available).	√
Priority	Tapping a job brings up a form allowing the priority to be edited.	√
SOE	Tapping 'Trigger SOE' throws an alert to the user.	√
	Data are refreshed after 'Trigger SOE' is confirmed.	√

6.6 Concluding remarks on Chapter 6

In this chapter, the mobile client of the reactive scheduling system was presented. The roles it fulfils in capturing external disturbances, e.g. new jobs, and as data support

platform were discussed. A brief overview of a prominent mobile architecture, MVC was given. The UML process was applied to the design of the application. Certian aspects of the applications development were discussed.

By developing these requirement, and providing the data on a mobile device, the system state could be accessed on-the-spot giving floor managers insight into the system they otherwise would not have been able to access. In this way, floor managers could be proactive about the delivery of quality products in a timely and cost-effective manner.

Chapter 7

Development of the reactive scheduling application

Management of both the physical components of a reactive scheduling system (RSS), and the coordination of its software components (and the data these components exchange) is a role that must largely be assumed by a suite of back-end services. These services, in addition to performing dedicated back-end activities, must also support the requirements of the front-end devices, e.g. sensor network nodes capturing work-in-process (WIP) movement. The architecture of such a back-end system not only supports the operational capabilities of the system, but also imposes constraints on the manner in which the system can expand, i.e., its potential to scale. This is an important factor within the manufacturing context as the structural configuration and logical operation of these systems, such as the pressure gauge assembly operation described in **Chapter 3**, are likely to change over time due to, for example, demand fluctuations, the introduction of new products, or advancements in manufacturing technologies. Since the effectiveness of an RSS is dependent on it possessing an adequate model of the system, the design of the back-end should allow the model to be updated, without necessarily updating the front end components.

The construction of a back-end system capable of supporting both the functional requirements of the front-end devices and the back-end requirements of an RSS, and furthermore, one which conforms to an architecture with desirables properties, is the topic of this chapter.

7.1 Requirements of the reactive scheduling application

Generic business requirements for a reactive scheduling application (RSA) were set out in Section 2.4.4, these are repeated here:

1. Manage system configuration,
2. Update the snapshot of the shop floor as new disturbances arise,
3. Provide an interface for the data acquisition and data visualisation components,
4. Process incoming events in accordance with the rescheduling policy,
5. Host the simulation model of the shop floor,
6. Invoke the simulation experiment and interpret the simulation model's output, and
7. Convert the model's output into meaningful control instructions.

Furthermore, in **Chapter 5** and **Chapter 6**, requirements for the front-end devices (specifically related to the industry partner) were gathered and documented. These front end requirement were used to drive the feature set of the RSA.

7.2 Designing the backend system

The primary design goal of the RSA was to allow the SOE to be offered to the shop floor as an independent service (see 'service tier' in Figure 7.1). In this way the service could be made agnostic to any client implementation. Thus as the technology within the sensor network and mobile spaces advance, incorporation of these new technologies into the RSS would be possible without needing server-side changes (apart from configuration).

During the inception phase, when the project vision was created, it was recognised that an back-end architecture would be needed to guide the creation of the business tier. Thus, prior to starting the design and analysis workflows of the elaboration phase, an architecture suitable for the back-end of the RSS needed to be found.

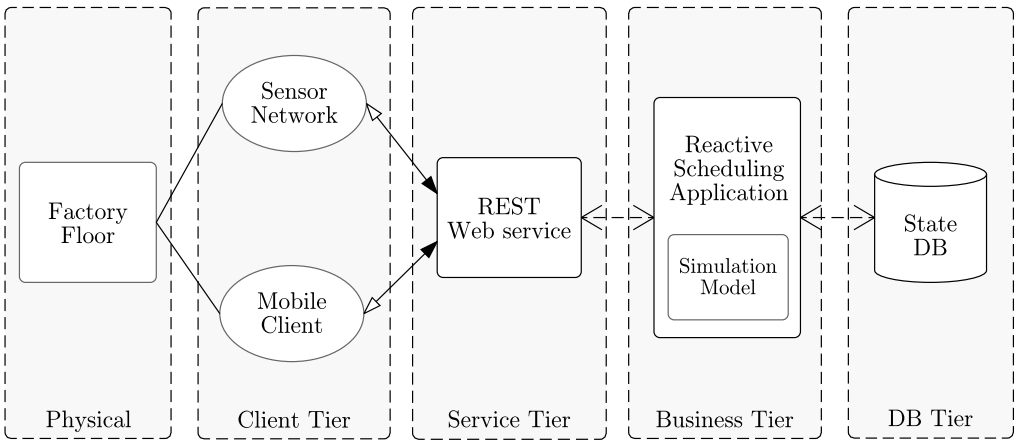


Figure 7.1: Tiered view of the reactive scheduling system.

7.2.1 Service-orientated software architecture

The architectural decision to offer the reactive scheduling application (RSA) as an internet-based service to the shop floor invited the question how geographically distribute components should be connected over the internet. Research into the field revealed that Web services were the industry standard method for connecting network separated software systems. To assist in designing a Web service, the service orientated architecture (SOA) was selected. SOA is a software architecture philosophy in which components are designed to expose services to other components through an agreed upon communication protocol. According to Erl (2008), a service acts as a container of related capabilities where each capability is the product of the logic of internal components. Services communicate with service consumers through a *service contract*. This contract defines the public facing logic, or capabilities, the component is seeking to expose. Erl (2008) identifies a number of tenets of SOA, including:

- Standardised service contract: Network separated components must agree upon a well defined consistent service contract. All interaction with capabilities of an application must happen through the service interface and it the manner defined by the service document.
- Loose coupling: The degree to which software components depend on each other must be minimised. For example, changes to application application logic should

not affect the service interface, i.e. break the service contract.

- Statelessness: State management should not extend between the request-response cycle. In addition to consuming resources, state management can hamper reusability and scalability. Services must be designed intelligently to negate the overhead of state management.

Other tenets include: abstraction, anatomy, and statelessness, further discussed in Section 7.3.2. Achieving a SOA involved the application of the tenets of service-orientated design paradigm as well as a central concept within SOE, the of a separation of concerns (Erl, 2008). Essentially, this involves the deconstruction of the problem into hierarchical collection of sub problems, or concerns; allowing the solution to be arrived at more effectively.

Selection of software architecture ensured analysis and design could progress according to some plan, and brought the elaboration phase in line with the 'architecture centric' element of the unified process (Section 4.2.2).

7.2.2 Applying the UML modelling process

The first goal of the elaboration phase was to partition the application into components. To achieve this, a high level logical view of the processes of the system and the data they would consume was created using data flow diagrams (DFDs). The resulting context and level one diagrams are shown in Figure E.8 and Figure E.9 respectively. This step aided in aggregating functions into processes, and in beginning to understand were service interfaces would need to be placed. After creating the level one DFD, the logical processes were analysed and allocated to components. The top level component view is shown in Figure E.3. These components defined the physical partitioning of the source code, and importantly defined the service interfaces between the components. To better understand the lifecyle of the RSA statechart diagrams were created. Figure E.11 shown an example of a state chart diagram used to captured the behaviour of the system during event processing and SOE invocation.

After examination of the system at a high level was complete, components were further refined. A summary of architectural decision made each is given below:

- **Data manager**: The primary purpose of the data manager was to handle data synchronisation. That is, as events arrived from the shop floor, the data manager would update the system snapshot. This involved, for example, transitioning WIP

items through their sequence of workstation queues (termed an event chain). Secondary functions included; refreshing the WIP data the simulation model would require for initialisation, and providing helper classes to other objects seeking to interact with the database. The internal components of the data manager are shown in Figure E.6.

Design of the queue manager was assisted by the creation of statechart diagrams. These diagrams helped in understanding the state progression of entities in the assembly operation, e.g. 'job' and 'task' (Figure E.12 and Figure E.13).

- **SOE**: The simulation optimisation engine (SOE) was designed as a wrapper around the simulation experiment. The architectural focus when designing the SOE was to encapsulate all the complexity involved in updating, running, and interpreting a simulation experiment. In doing so, the SOE provided an abstracted interface through with other objects could interact with the experiment. The logical flow of the SOE was established using a data flow diagram, see Figure E.10, which allowed the internal components of the SOE to be designed (see Figure E.7). Figure D.5 shows a timing diagrams for manually triggering SOE.

- **Coordinator**: As shown in Figure E.4, the coordinator was designed to fulfil two roles. First to processes all incoming events from the Web service, and secondly to check if events were white-listed. The 'policy setter' component exposed an interface (represented by round circle) to the event filter (half circle) consuming this service.

- **Database**: A persistent store in the form of a relational database was created to support the RSA by storing 1) configuration information, e.g. RFID—task associations, 2) snapshot information, e.g WIP positions, and 3) data generated by the simulation model, e.g job expected completion times. An entity relationship diagram of a sub-set of the tables in the database is shown in Figure E.17. This design meets the requirements set out in Section 2.4.5.

- **Web service**: The Web service was designed to have a set of controllers, each mapping to a resource that was to be exposed. Figure E.5 shows each of these controllers. Detail on exposing this Web service is given in Section 7.3.2. Sequence diagrams detailing the message flows between the Web service and front-end were shown in the preceding chapters.

The set of UML models presented above met the requirement for the elaboration phase of the unified process, i.e. a baseline architecture.

7.3 Constructing the RSA

To demonstrate the work performed, this section details the construction of two of the components of the RSA: the simulation optimisation engine and the client-facing Web service. Although construction largely executed against the architecture description produced during analysis and design (documented using UML diagrams), iterations through the construction phase raised many unforeseen issues which forced adjustments to the architecture—a process encouraged by the iterative nature of unified process.

7.3.1 Simulation optimisation engine

As discussed, the SOE was intended to abstract the complexity of interacting with the simulation experiment by encapsulating all experimentation logic and exposing a simplified interface to other objects, effectively 'wrapping' the simulation experiment. The SOE itself was the aggregation of a set of sub components, shown in Figure 7.2, each designed to perform a single task. By encapsulating experimentation complexity, objects, such as the coordinator, could easily invoke an experiment with a single message to the SOE object. A brief description of each of the SOE's sub components is provided below:

- Simio file IO: The file IO component was responsible for loading the simulation model from disk into memory.
- Experiment input analyser: Jobs whose release event had not yet been received, i.e. the job had not entered the assembly operation, had their release dates brought forward to match the simulation experiment's starting time by the experiment input analyser. Therefore preventing late jobs from being overlooked by the experiment. The experiment input analyser also determined the experiment's starting time by extracting the last processed event.
- Experiment runner: The experiment runner was responsible for the interaction between the RSA and the simulation model. This component integrated directly into the simulation software's application programming interface (API) and was tasked with invoking the experiment. Additionally, this component handled relevant call-backs from the experiment, e.g. run completed, scenario completed. These call-backs were used to gather sample outputs.
- Scenario comparator: The implementation of the Kim Nelson procedure was the primary purpose of the scenario comparator component. By executing the Kim Nelson procedure, the comparator was able to select the best scenario with

respect to the chosen performance measure. This scenario was then passed to the experiment output analyser for analysis. Additional functionality such as selecting a random scenario and selecting a scenario by name were added for debugging purposes.

- Experiment output analyser: After extracting the dispatching rule from the received scenario, the experiment output analyser was responsible for performing dynamic rank assignment. All tasks in each of the monitored workstation's input buffers were updated, after which each of the job's expected completion time was updated in the database.

Definitions of the object interfaces of the components of the SOE are given in Appendix G.

7.3.1.1 Integration of the Kim Nelson procedure

The Kim Nelson procedure (Kim & Nelson, 2001) was used as a statistical means of selecting the scenario exhibiting the best response to a dispatching rule. Here 'best' is defined as the maximum or minimum of the performance measure of the system (performance measure of the gauge assembly operation were defined in Section 3.4). Incorporating this procedure was necessary as the simulation model of **Chapter 3** was not programmed to include any means of interpreting or comparing the effect each dispatching rule had on the simulation model. The task of comparing the scenarios was therefore including in the logic of the SOE which interacted directly with the simulation model.

The Kim Nelson procedure has two useful attributes, first, ensuring the statistically best scenario of the k finite available scenarios is selected, and secondly, efficiently handling the number of replications that are run, thus reducing the computational burden. The second point is due to the fact that the procedure systematically discards unworthy scenarios as the algorithm advances. This is in contrast to other procedures which allocate an identical number of replications to each scenario. This policy can be inefficient as certain scenarios may exhibit low variance and therefore do not require all allocated replications to sufficiently narrow the confidence interval.

Essentially, the procedure works as follows:

1. At the end of each replication (across each scenario set) scenario outputs are evaluated with respect to all other scenarios after which statistically inferior scenarios are removed from the evaluated set.

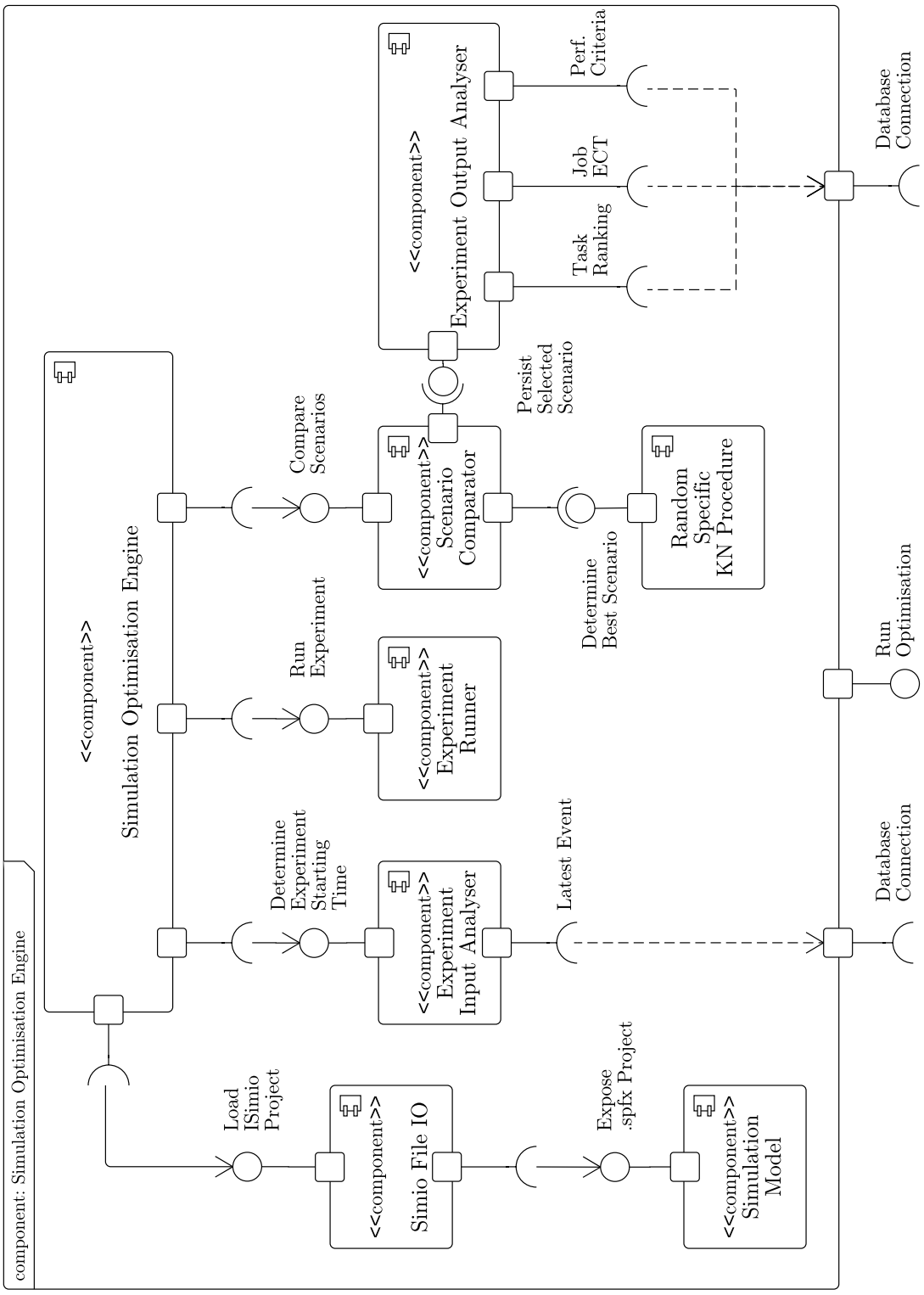


Figure 7.2: Simulation optimisation engine.

2. An additional replication is then assigned to each of the remaining scenarios.

3. This procedure is repeated until there is a single scenario remaining. This scenario is then deemed the 'best' scenario.

The procedure assures with a confidence level of $1-\alpha$ that the true mean of the best scenario will have a difference of at least δ (termed the indifference zone) to the true mean of the second best scenario. The indifference zone is a parameter specified by the experimenter which represents a practically significant difference worth detecting Kim & Nelson (2001). Formally, the procedure can be described mathematically as follows Kim & Nelson (2001):

1. Experiment Setup

   Input Parameters (level of the experiment):

   Confidence level $1-\alpha$.

   Indifference zone δ.

   Maximum number of replications across each scenario N_M

   First stage sample size $n_0 \geq 2$.

2. Initialisation

   Let $I = 1, 2, \ldots, k$ be the set of scenarios still in contention, and let $h^2 = 2_{c\eta} \times (n_0 - 1)$.

   Obtain n_0 observations $\hat{Y}_{ij}, j = 1, 2, \ldots, n_0$ from each scenario $i = 1, 2, \ldots, k$. For all $i \neq l$ compute

$$S_{il}^2 = \frac{1}{n_0 - 1} \sum_{j=1}^{n_0} \left(\hat{Y}_{ij} - \hat{Y}_{lj} - \left[\bar{\hat{Y}}_i(n_0) - \bar{\hat{Y}}_l(n_0) \right] \right)^2 \tag{7.1}$$

   the sample variance of the difference between scenarios i and l. Let

$$N_{il} = \left\lceil \frac{h^2 S_{il}^2}{\delta^2} \right\rceil, \tag{7.2}$$

   where [.] indicates truncation of any fractional part, let

$$N_i = \max_{l \neq i} N_{il}. \tag{7.3}$$

   Here $N_i + 1$ is the maximum number of observations that can be taken from scenario i. If $n_0 > \max_i N_i$, then stop and select the scenario with the largest $\bar{\hat{Y}}_i(n_0)$ as the best. Otherwise, set the observation counter $r = n_0$ and go to screening.

3. Screening
   For objective maximisation, set $I^{\text{old}} = I$. Let

$$I = \{i\colon i \in I^{\text{old}} \text{ and } \bar{\hat{Y}}_i(r) \geq \bar{\hat{Y}}_l(r) + W_{il}(r), \forall l \in I^{\text{old}}, l \neq i\}, \tag{7.4}$$

   For objective minimisation, set $I^{\text{old}} = I$. Let

$$I = \{i\colon i \in I^{\text{old}} \text{ and } \bar{\hat{Y}}_i(r) \leq \bar{\hat{Y}}_l(r) - W_{il}(r), \forall l \in I^{\text{old}}, l \neq i\}, \tag{7.5}$$

   where the whisker is given by

$$W_{il} = max\left\{0, \frac{\delta}{2cr}\left(\frac{h^2 S_{il}^2}{\delta^2} - r\right)\right\} \tag{7.6}$$

   With respect to maximisation, $W_{il}(r)$ represents the maximum distance below scenario l within which the sample mean from scenario i must reside or else face elimination. $W_{il}(r)$ is evaluated for all scenario combinations.

4. Stopping condition If $|I| = 1$, stop, select the system remaining system in I as the best.

5. Constants

   The constant η is the solution to the equation

$$g(\eta) \equiv \sum_{l=1}^{c}(-1)^{l+1}\left(1 - \frac{1}{2}\mathcal{I}(l=c)\right)\left(1 + \frac{2\eta(2c-l)l}{c}\right)^{-(n_0-1)/2} \tag{7.7}$$

   where $\mathcal{I}$ is the indicator function. The constant c may be any non-negative integer, however in the special case that $c = 1$, the closed-form solution becomes:

$$\eta = \frac{1}{2}\left[\left(\frac{2\alpha}{k-1}\right)^{-2/(n_0-1)} - 1\right] \tag{7.8}$$

The implementation of this procedure demanded that the interaction between the scenario comparator component and the simulation experiment's API be constructed in such a way as to allow complete control of the experiment and the continuous sampling of replication. This involved manipulating simulation software's API and creating a set of custom mapping objects to extract data at the end of each replication (as opposed to at the end of a scenario) enabling the fully sequential procedure. Naturally, the output data of the scenario returned by the Kim Nelson procedure was used to update the system metrics and to perform task ranking.

7.3.1.2 Dynamic coupling and state initialisation

The specifics around how the RSA should be integrated into the simulation model were not immediately clear during the elaboration phase. Since little documentation existed detailing how to approach the integration (especially with regard to extracting non-standard metrics such as job estimated completion times), it was decided that a prototype including 'mock' simulation model would need to first be built. The mock model was designed to mimic the interface and the input and output data that would be exchanged using true simulation model. This simplified model assisted in both defining the order of interaction between the SOE and the model, and also in dynamically coupling the SOE to the simulation experiment. As the number of entities in the model was potentially different at each rescheduling point (e.g. jobs in the system), the data structures needed to adapted to the variable length and quantity of the experiments outputs. output data.

Prior to running the simulation model, the latest snapshot of the shop floor, containing for example WIP positions, needed to be used to initialise the simulation model. To achieve this a data-pull mechanism was implemented using a custom data binder. This binder was used to link the set relational tables (used by the simulation model to initialise itself) to a set of tables within the RSA's state database. Furthermore, the RSA state database itself also needed to be initialised in a state matching the operational phase. Test data were stored in a configuration side-car file and inserted into the database base at application compile time.

7.3.2 Exposing the RSA through a Web service

As discussed in Section 7.2, one of the primary design goals was to offer the RSA as a software service to the shop floor. Realising this goal involved the construction of a Web service to expose certain capabilities of the RSA over the internet. In doing so, front-end devices would be able to consume the service through any internet connection. Web services, according to Gottschalk *et al.* (2002), define an interface, accessible through an end-point, that describes a collection of operations which are network-accessible through a standardised messaging protocol. Furthermore, Chappell (2004) defines the purpose of a Web service as being to provide a service abstraction enabling interoperability between applications running on different platforms and devices.

By constructing a Web service[1], certain back-end capabilities could be exposed over the internet, allowing the front-end devices on the shop floor to interact with the RSA

[1]The advantages of Web services are listed in Section E.1.2.

server by consuming this service. Put another way, the Web service, by way of a service contract, became the external interface, or API façade, of the RSA.

7.3.2.1 Incorporating the RESTful architectural style

The selection of a Web service architectural style followed both a qualitative investigation into the properties of Web services, and research into many quantitative aspects. The objective of this research was to identify a means of integrating the front-end devices with the back-end application server whilst permitting the interface to the shop floor to be interoperable, i.e. language and environment neutral.

The RESTful architectural style[1] was identified as the most suitable architecture for this purpose. In addition to the properties and advantages of this architecture (described in detail in the sections below), REST's popularity in the modern Web, and the relative simplicity of publishing and consuming REST web services, led to its selection of over other Web service architectures such as XML-RPC/(SOAP) .

The term representational state transfer (REST) was introduced by Fielding (2000) as a modern architectural style for designing Web services. REST currently serves as the underlying architectural style for most of the modern Web. REST, broadly speaking, places a set of restrictions on how resources (concepts relating to data) should be identified and represented when accessed via a Web service. Operations on resources are performed by making explicit use of the standard HTTP verbs: GET, POST, PUT, and DELETE. These verbs correlate to the traditional 'create', 'read', 'update', and 'delete' operations—though the mapping is not strictly one-to-one. Furthermore, interaction between the client and the server must be stateless, meaning the server should not maintain state *between* requests (Fowler, 2003). Due to its reliance on the HTTP protocol, REST offers compatibility with almost any client capable of performing an HTTP request.

REST services, by design, promote service-orientated architectures, and furthermore have a number of desirable properties for distributed systems (such as the RSS developed in this thesis). Those properties worth highlighting are (Fielding, 2000):

- Emphasis on scalability of component interactions,

[1] A full comparison of the Web service landscape and the architectures that produce them is beyond the scope of this thesis. However, the reader is directed to Gottschalk *et al.* (2002) for an introduction to the Web service landscape and to Pautasso *et al.* (2008) for an in-depth quantitative technical comparison between REST and other popular RPC (or 'Big' Web service) based architectures.

- Generality of interfaces,
- Straightforward API publication and documentation, and
- Endpoints may be easily debugged using packet observers.

Six architectural constrains, set out by Fielding (2000), needed to be adhered to for a Web service to be deemed RESTful, they are as follows:

1. Client-server: A clear separation of concerns is required. For example, clients should not be concerned with data storage. This improves portability across multiple platforms and scalability by reducing client complexity.
2. Stateless: Every HTTP request happens in complete isolation. All necessary information is included in the request, i.e, the server should never maintain state between requests.
3. Uniform interface: A well defined uniform interface between components is required—encouraging independent evolvability.
4. Layered system: Intermediary components may improve system scalability, e.g., separation of business logic and the data model, load balancing, shared caching, etc.
5. Cache-able: Responses must be cache-able, potentially eliminating redundant requests (see Section D.1.4).

Three of the five required constraints listed above were implemented during Web service development. The requirements of 'layered system' (see Section E.1.4) and 'cacheble' (see Section D.1.4) were not implemented due to time constraints. The RFC 2616 HTTP/1.1 specification by Fielding *et al.* (1999) served as a reference during the implementation of the Web service[1].

7.3.2.2 Defining the service contract

The Web service was intended to facilitate the functions of both the data acquisition and the decision support platforms. Its design, therefore, was driven by the use case diagrams of the sensor network and mobile client—discussed in Sections 5.4 and 6.1 respectively. Definition of the API occurred during the elaboration phases of both the

[1]This specification has since been updated to the specifications set RFC 723X HTTP/1.1 (Fielding *et al.*, 2014)

front-end and back-end components, though further adjustments were made during construction as more domain knowledge on REST was gathered. The following steps were followed in arriving at the service contract based on the REST style (documented in Appendix F).

1. Resource identification: A list of resources front-end devices required to interact with was compiled. These resources did not necessarily map to database entities but rather were abstractions of the underlying data model. The identified resources were: job, task, card, operation, and SOE.

2. URI assignment: Each resource was assigned a URI. This URI to allowed service consumers to locate the resource on the server, e.g. `http://<#URL#>/api/job/<job-Id>/priority`.

3. Actions: The action to be performed on the resource was defined and an appropriate HTTP verb selected, e.g., for the action 'fetch a list of jobs' the HTTP GET verb was selected. GET is classed as idempotent as no modification to the *state* of the resource is made[1].

4. Resource representations: Stateless representation of each of the identified resources were defined in JSON format. JSON objects served as semantic blueprints of the resource representations.

5. Documentation: The API was documented to formalise the service contract and aid in the integration effort between the front-end and the Web service.

The API is an example of a service contract in the system, agreed upon between the RSA and the front-end devices (see Section 7.2.1).

7.3.2.3 Development of the API

In accordance with service-orientated architecture and REST principles, the Web service layer and data layer were separated. Tailored data representations were created specifically for the consumption by clients. In other words, the representation of the resources exposed through the API did not necessarily match the underlying database representation. This principle is demonstrated in Figure 7.3.

[1] A notable deviation from the REST style was in the use of the GET verb for the logging of WIP transitions (by the sensor network). Since accessing the WIP resource causes a state change on the server, a POST would have been more appropriate—however client-side software limitations prevented its implementation (see Section F.3).

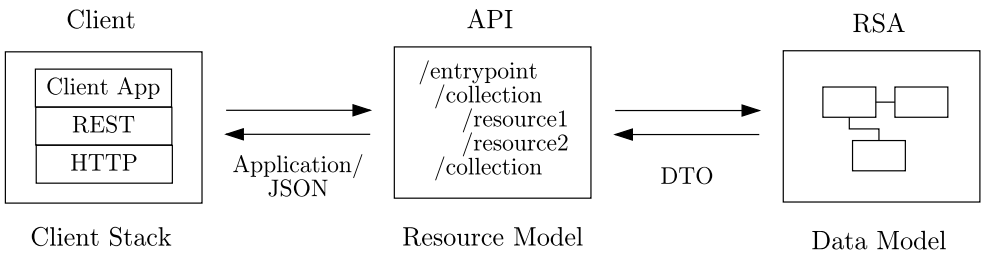


Figure 7.3: Three-tiered Web service model.

The challenge of exposing resources to clients while maintaining a loose service coupling was addressed with the use of data transfer objects (DTOs). DTOs assisted in creating a loosely coupled service by adding a data abstraction layer above the database. This meant that the structure, or the even technology, of the database could be changed without the need to rewrite the Web service. Web service controllers accessed representations of resources, e.g. job, through DTO objects, making the service independent of the way the resource was persisted in the data model. Similarly, DTO's could be adjusted to meet a new data requirement on the front-end without needing to change the structure of the data model.

Figure 7.3 also illustrates the purpose of the service contract as defining the communication between the two systems. The business logic and data model of the RSA were bound to support the DTOs. Whereas front-end devices were bound to the definition of the API. This is one example of the layering reactive scheduling system to reduce its complexity.

7.3.3 Management portal

A management portal was created allow the administrator of the system, typically individuals in production planning and control, to manage the physical devices of the system. Screenshots are shown in Section E.4. The portal allowed users to create, edit, and delete sensor network nodes and RFID cards. Once added these device became available within the system for use, for example, once an RFID card was added, new tasks (set to be released onto the shop floor) could be linked with the card for tracking purposes.

7.4 Deployment of the RSS

The deployment configuration of the RSS is shown in Figure E.14 (**Appendix** E). The purpose of the deployment diagram was to illustrate the relationship between the physical or conceptual targets of the solution and the informational assets assigned to them. Targets are shown as a perspective cubes (e.g. iOS), and software artifacts as rectangles (e.g. iOS app bundle). Features of the diagram are summarised below:

1. The mobile client's packaged app-bundle was deployed to the Apple's iOS operating system. The compositional relationship indicates the lifecyle the application was dependent on the physical device.

2. Compiled source code for the Waspmote was deployed to the Waspmote microcontroller's bootloader.

3. The reactive scheduling application, simulation model, and relational database were all deployed within a single solution onto a cloud-based virtual machine. The network architecture of the deployed solution within the cloud is shown in Figure E.15. An improvement to architecture could be made by dynamically loading the simulation model from an external system, rather than including it within the RSA's execution environment. This arrangement would allow the simulation model to updated and deployed independently of the RSA—negating the need to redeploy the whole back-end system.

Microsoft's Azure^{TM} cloud platform was used as the deployment environment for the RSA and its supporting database. Geographically, the application was hosted within a cloud data centre was based in Western Europe. This region was selected as it exhibited good latency values, though variation between regions was found to be low. Deployment to the cloud occurred throughout the development, this made incremental release and testing possible cycle of the unified process possible.

7.5 System integration testing

In addition to testing the functionality of the RSA, the interactions between each of the systems needed to tested. The procedure included: 1) ensuring captured system disturbances were hitting Web service endpoints 2) checking that request date formulated by the front-end devices conformed to the service contract 3) verifying that incoming data from the Web service had in fact been persisted to the database layer 4) ensuring that

resource representations returned by the front-end devices matched the representations formulated by the application DTOs.

Table 7.1: RSA system integration testing.

Component	Criteria	Validated
Simio File IO	Successfully locates the Simio file when executed on both local and cloud server.	√
	Loads Simio project file into memory.	√
Experiment Input Analyser	Experiment starting time is correctly calculated even if no events have been received.	√
Experiment Output Analyser	Job expected completion times are extracted from the CMExperiment correctly.	√
	The correct task ranking is performed according to the selected scenario.	√
Experiment Runner	Successfully invokes a simulation experiment.	√
	Creates a CMExperiment from pivot grid data, including job completion times.	√
Scenario Comparator	Kim Nelson procedure is successfully invoked.	*
	Scenario selected by custom KN procedure matches scenario selected by Simio KN.	√
Coordinator	Coordinator is notified when internal and external disturbances occur, including, WIP event, job added, job priority change, machine failure.	√
SOE	Successfully invokes the scheduling optimisation engine.	√
	Release times of stagnant jobs are updated.	√
	Task rankings and Job expected completion times are persisted to the database after the best scenario has been selected.	√
Mobile Client	Task rankings returned by the GET request match the rankings in the database.	√
	New job data matches the data sent when adding a job via a POST request.	√
	On device performance metrics match those on the server after a GET request has been performed.	√

Continued on next page

Component	Criteria	Validated
Sensor Network	Captured transactions are received by the server. The data parsed by the Web service matches that sent by the Node.	√
API	RESTful endpoints are available to clients over the internet.	√
	Endpoints address the correct resources on the server.	√
	GET, PUT, and DELETE request are idempotent, i.e., the state of the server remains the same if the same request is issued once, or multiple times.	√
Simulation Model	New jobs that appear in the relational tables are released into the simulation model at the release date instant.	√
	ADOGridDataProvider imports experiment initialisation data and inserts the data into the simulation model relational tables.	*
	Only the latest WIP event in each task's event chain is used to instantiate a WIP item in the Simulation model.	√

7.6 Concluding remarks on Chapter 7

In this chapter the design and construction of the reactive scheduling application (RSA) was described. A service orientated architecture was introduced as the base architecture model for the RSA where after the UML modelling process was applied using this architectural pattern. Two components of the RSA were highlighted: the simulation optimisation engine and the Web service. The function of each of the internal components of the SOE was described and the implementation of the Kim Nelson procedure (Kim & Nelson, 2001) was detailed. Web services were introduced as a means of exposing application functionality over the internet and the specification of a Web service contract was discussed. The cloud deployment of the reactive scheduling system was summarised, including the current cloud network architecture.

Chapter 8

Experimentation and results

The complexity inherent in software-intensive systems, such as the a reactive scheduling system (RSS) developed in this thesis, make it difficult to express architectural decisions through source code structure alone. Architecture description languages, such as the unified modelling language (UML), assist in standardising the visualisation of complex systems. Throughout the preceding chapters a combination of textual descriptions, conceptual diagrams, and UML diagrams were provided to communicate the architectural design points. The purpose of the architecture within the context of the project to develop an RSS was to specify the components of the system from multiple abstraction levels and from different viewpoints (e.g logical, physical, deployment, etc.). The resulting architecture description was used as an input to the construction phase of the system.

8.1 Summary of the proposed architecture

The conceptual diagram shown in Figure 8.1 is the unification of all the RSS components constructed in this thesis. Each component is shown at an abstracted level of detail with the focus being on its logical separation and communication points. Furthermore, the diagram documents the culmination of the application of the iterative unified process life-cycle model. Additionally, it represents the outcome of the UML modelling process, and the variety of architectural styles (e.g. RESTful), design patterns (e.g. model-view-controller, object-orientated), and programming principles (e.g. DRY) used throughout the RSS's construction. Some of the key features are listed below:

- Database layer: The database layer was used to persist all relevant information consumed and generated by the system. The main objective of database was to

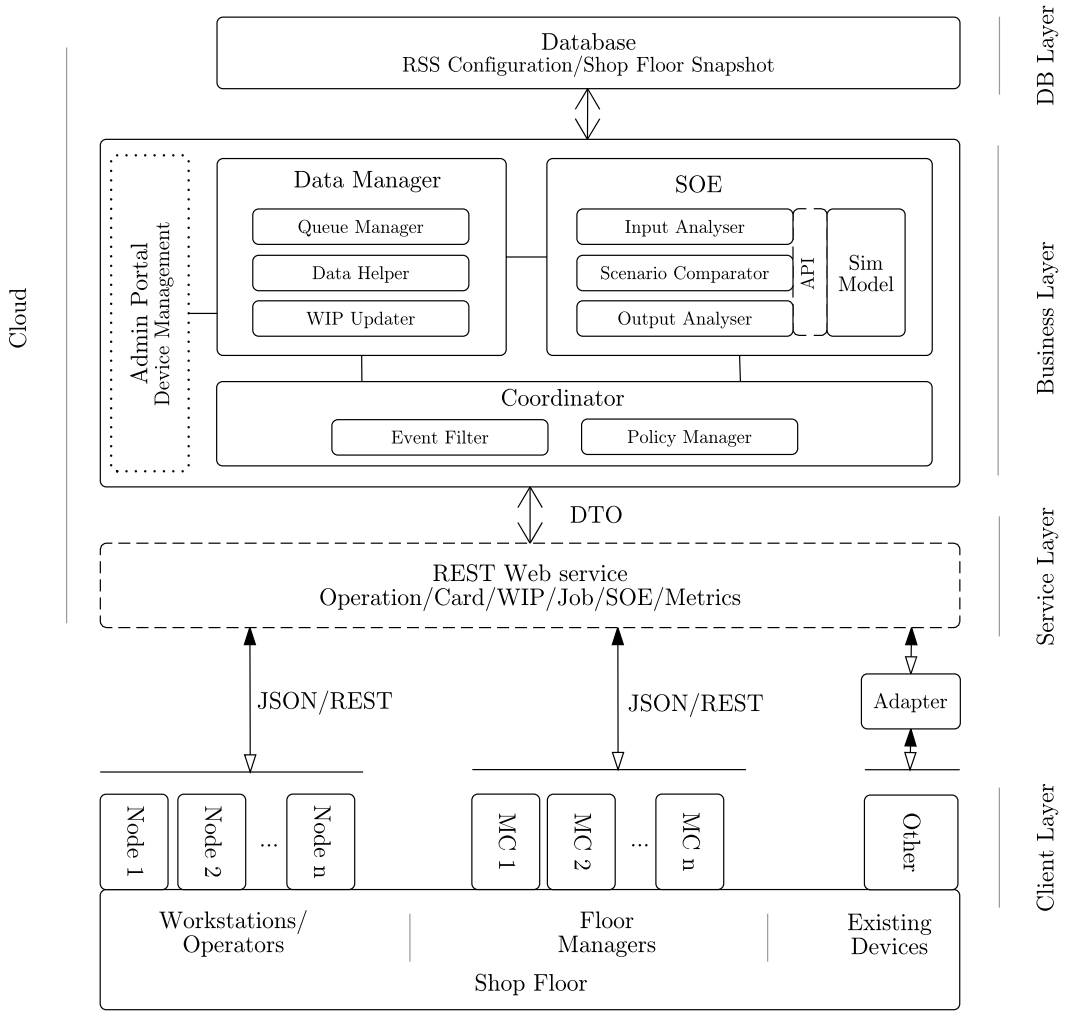


Figure 8.1: Summary of the proposed architecture for an RSS.

store a snapshot of the factory floor and allow it to be updated as time progressed. This snapshot included both the imminent jobs set to be released onto the shop floor (captured by the mobile client) and the position of WIP items (captured by sensor network nodes). To support this objective the database managed configuration information, for example, associations between tasks and RFID cards, nodes and workstations, etc.

- Business layer: This layer was responsible for first, updating the system snapshot as new data arrived through the Web service interface from the shop floor. Secondly, for observing incoming events (e.g. new orders or WIP transitions) and then triggering the scheduling engine when the rescheduling policy permitted it. Thirdly, for performing a statistical comparison between dispatching rule scenarios (using a simulation model which adequately represented the operation of the shop floor). Fourthly, using the selected dispatching rule to update workstation input buffer rankings.

- Service layer: The business layer exposed certain capabilities over the internet using a RESTful styled Web service. This essentially allowed any network attached client to discover and consume the service (via a URI). In keeping with service-orientated architecture (SOA) requirements, a service contract was drawn up with the purpose of defining the protocol and semantics of the resource-based exchanges between client and the server (business layer).

- Client layer: The client layer comprised a set of sensor network nodes and a mobile client. These devices were deployed to the factory floor and were used by floor managers and workstation operators to interface with the RSA. The sensor network, designed as a data acquisition platform for internal disturbances, captured WIP transitions with the help of workstation operators. The mobile client acted as both a data acquisition platform for external disturbances and a decision support platform to aid floor managers by giving them insight into the system, e.g. view job estimated completion times.

Needless to say, an architecture is vital when designing large complex software-intensive systems, such as the one proposed above. In addition to having the necessary time and resources, upfront planning of an architecture may define the line between success and failure. This is because, by defining an architecture at an early stage of a project, its decisions directly influence many other decision through the phases of the project, right through till its eventual construction. At this point the properties, principles and techniques that comprise a good software architecture have been discussed,

and an overview of the RSS architecture has been given. The next section deals with the evaluation of the architecture according to a few attributes Booch (2007) describes central to an architectures success.

8.2 Evaluation of the proposed architecture

Booch (2007) identifies key attributes common to all successful software architectures, and suggests that for newly proposed systems, these attributes be present. Examples revealing these attributes within the proposed architecture are documented below.

8.2.1 Levels of abstraction

Software architecture must exhibit well-defined layers of abstraction. These layers assist in understanding the system at manageable levels of complexity. Within the documented UML architecture description, abstraction is common. Naturally, all diagrams are themselves abstractions of the run-time elements of the software system, however it is the layering of abstraction that is the key point for an architecture description. Notable examples for layering used in this thesis include, firstly, data flow diagrams which incrementally specify the processes of the system at incrementally lower level of abstraction (see Figures C.5, C.6, and 5.5. And secondly, component diagrams, for example Figure E.3 which shows the RSS components at the highest level of abstraction. The component view was then decomposed into four individual components diagram at a lower level of abstraction (see Figures E.4, E.5, E.6, and E.7).

The layers of abstraction contained within these UML diagrams were adhered to during the construction phase. The most prominent example of run-time abstraction is the simulation optimisation engine. This component, in keeping with its design, provided a simple interface, with a single invocation method to other objects. Thus, objects interesting in running the simulation experiment (e.g. the coordinator) need only call into a single method on the SOE. This simple interface abstracted all of the SOE's internal logic (see component interfaces in Appendix G), i.e., loading a file from disk, setting up the experiment, running the experiment, etc. In fact, the simulation model within the SOE is a further example of abstraction as it provides a programmatic interface masking the internal complexity of the simulation model.

8.2.2 Separation of concerns

A key object-orientated principle that assisted in enforcing a separation of concerns was the single responsibility principle. The principle states that classes and components

should be given ownership over one part of functionality provided system. The design of the RSS backend system is a concrete example of the separation of concerns. The Web service (or interface of the application) was decoupled from the business layer to allow the service contract to be defined independently of the application logic. For example, multiple Web service endpoints (described in Appendix F) were designed to expose functionality related to a single resource, e.g. job or RFID card. This is in contrast to say, a design in which *all* functionality is exposed through a single endpoint returning a single global resource. In this comparatively bad design, multiple concerns around a single resources are exposed, requiring the client to operate on only an isolated portion.

The design of the mobile client is also a good example of the separation of concerns at the functional level. The principles of the model-view-controller design pattern ensured a clear separation between the user interface elements, controllers (which handled business logic), and models (responsible for data persistence). The benefit of this approach was that changes to the implementation of one layer could be made without affecting the other layers (granted the change did not affect the layer's interface). In designing the sensor network nodes, concerns were identified and separated, for example, the processes of awaiting input from state transition buttons, reading a RFID card's unique identifier, and communicating to the Web service through a WiFi network were each separated into logical units (see Figure 5.5).

8.2.3 Architecture quality attributes

The potential real-world applicability of the architecture can be demonstrated by highlighting the its conformance to key software quality attributes (as defined by IEEE standard 610.12-1990 (IEEE, 1999)).

- Maintainability: The proposed architecture was designed to be maintainable within the context of the system in which it operated. Maintainability is not only a function of good architecture, but also of good coding practices. The principles of separation of concerns and object orientated programming assist in making objects, and the modules they make up, maintainable. This is due mainly to the principle of encapsulation where all implementation details are hidden from the public scope. Only those methods and properties necessary for public consumption are exposed. Thus, modifying or updating the implementation of a class is possible without needing to modify the calling classes (granted the interface remains unchanged).

  The extent to which the architecture is maintainable is best explored by considering what would need to be changed if the architecture we migrated to a similar job-shop scheduling environment, i.e. one in which entities are both processed

at workstations and advance through set fixed sequences. In such a case, many components would need only moderate modification to be reused. For example, the SOE has the potential to be retrofitted for a second scheduler as its logic for updating, running, and interpreting a simulation experiment is neatly encapsulated and abstracted through an interface. The integration between the SOE and a second simulation model is possible, but is subject to a three conditions. First, the model would need to be built using the same simulation software (Simio®), secondly, it would need to conform to the same experiment configuration (i.e. a single dispatching rule per scenario), and thirdly, all non-standard responses (e.g. job completion times) would need to be encoded as custom output statistics.

- Scalability: Scaling of the frontend of the RSS is possible predominantly due to the architectural decision to place a Web service between the back-end. Adding additional nodes and clients to the system did not require a modification to the simulation model or to the other components of the RSA. This fact allowed for the front-end to scale horizontally. Similarly, the removal of devices was permissible and did not cause system failure. For example, the loss of a node due to a technical failure translated into a loss of visibility at its associated workstation. In other words, WIP transitions would not be communicated to the RSA and therefore WIP initialisation would not occur.

  In the case where no nodes are deployed to the shop floor the SOE would only considered those jobs set for imminent release.

- Loose coupling: By defining a service contract that specified a communication protocol and allowable semantics, both the business layer and client layer were forced to respect the contract in order to communicate (see Figure 8.1). In contrast, had the business layer been permitted to modify the protocol or adjust the semantics (for example, by switching the HTTP media type from 'application/JSON' to 'text/html') between requests, clients would have needed to familiar with the internal logic of the RSA—resulting in a tight coupling between the systems. In addition to the Web service acting as a communication broker, loose coupling was further enforced by designing simple payload-based exchanges, i.e. swapping resources representations (Section 7.3.2.2). This is in contrast to an RPC operation request style where arguments, not resources, are exchanged.

The integration of each of these attributes into a single architecture was a challenge, and yet resulted in a simpler unified design. Had these attributes been though of after-the-fact, their implementation would have far more difficult—if not impossible. By

possessing these attributes the architecture of the RSS is considered to be ready to endure the challenges of a real-world implementation and demands. Though, full scale deployment of the system was beyond the scope of this thesis, it is hoped that the work will taken even further in the future, through to deployment.

8.3 Experimentation with the simulation model

To demonstrate the scheduling capability of the system, experimentation was performed on the simulation model. The aim of the experiment was to explore the model's response to typically work loads—derived from the work load observed at the pressure gauge manufacturer (see Section B.1 of Appendix B). Three rescheduling profiles were constructed (as shown below) where each profile consisted of both imminent jobs set to be released onto the factory floor and WIP to be initialised at model runtime. The mixture of jobs in each profile consisted of 80% utility and 20% process gauge types as per the industry partner's yearly production split.

Identifier	New Jobs	WIP	Features
RP_1	33	21	Standard
RP_2	16	36	Two expedited orders
RP_3	5	45	Two large orders, both with relaxed due dates

The rescheduling data profiles were constructed to represent the transition of gauges through the factory floor in a single day. After the first rescheduling profile was constructed, RP_1, and loaded into the simulation model, the model was advanced by three hours and the location of each job noted. This data served as input into a second rescheduling profile, followed by a third. Three comprehensive rescheduling 'snapshots' where therefore available to the model off of which to initialise for experimentation. Additionally, the rescheduling data profiles contained future dated job releases, that is, when the model advanced the next business day, scheduled jobs were released into the factory—mimicking the real-world operation.

For the purposes of the demonstration of the RSS, a rescheduling window of two weeks in length was selected. This gave the model a myopic yet stable period over which to assess the performance of the schedule [1].

[1] An investigating into the length of the rescheduling window is beyond the scope of this thesis and is left for future research

Ideally, the testing of the simulation model should have been done in conjunction with the data capture devices, where each device uploaded data in real-time to the simulation model. However, due to the high data demands of the simulation model pre-populated rescheduling profiles were used to drive the experimentation process.

8.3.1 Experiment responses

A summary of the performance measures used for experimentation within the simulation model are given below. It was initially thought that the schedule makespan would be an important metric for the model, though further research revealed that schedule makespan is mostly applicable to off-line scheduling techniques and was therefore disregarded.

Measure	Description
$\bar{\hat{F}}$	Mean flow time
TP	Throughput rate
W	Mean WIP level

Each performance measure was encoded at the level of the job, e.g., mean flow time describes the average time interval a job spends within the model without accounting for past behaviour (that is, the metrics are forward facing and only consider the activities and delays after the rescheduling instant).

8.3.2 Applying the Kim Nelson procedure

Experimentation was conducted in two stages. First, the experiment was executed using a predetermined and fixed number of replications (using parameters shown in Table 8.1). This allowed for a comparison of the performance measures (presented later in the chapter).

Table 8.1: Summary of experimentation parameters.

Performance measure	Half-width h	Confidence level α
$\bar{\hat{F}}$	5	0.05
TP	0.5	0.05
W	3	0.05

Secondly, the experiment was run using the Kim Nelson (KN) procedure Kim & Nelson (2001) to demonstrate the selection of a single scenario and therefore its associated dispatching rule. As discussed, the Kim nelson procedure is an efficient mechanism for the selection of the 'best' scenario from a set of k available scenarios. Table 8.2 shows a summary of the KN parameters used to experiment with each performance measure. The indifference zone parameters were selected based on what was reasoned to be the smallest meaningful difference that would need to be detected.

Table 8.2: Summary of the KN experiment parameters.

Performance measure	Indifference zone δ	Confidence level α
$\bar{\bar{F}}$	0.5	0.05
TP	0.5	0.05
W	1	0.05

Worth mentioning is that the confidence level of the KN procedure (shown in Table 8.2) differs from the confidence level of the replication sample mean parameter. That is, the KN procedure states that the probability of correctly selecting the scenario with the largest true mean (where the true mean of the best is at least δ better than the second best) is greater than or equal to $1 - \alpha$ (as a result of the Bonferroni inequality). Whereas the sample mean confidence level indicates the interval within which the true population parameter is set to lie with a probability of $1 - \alpha$.

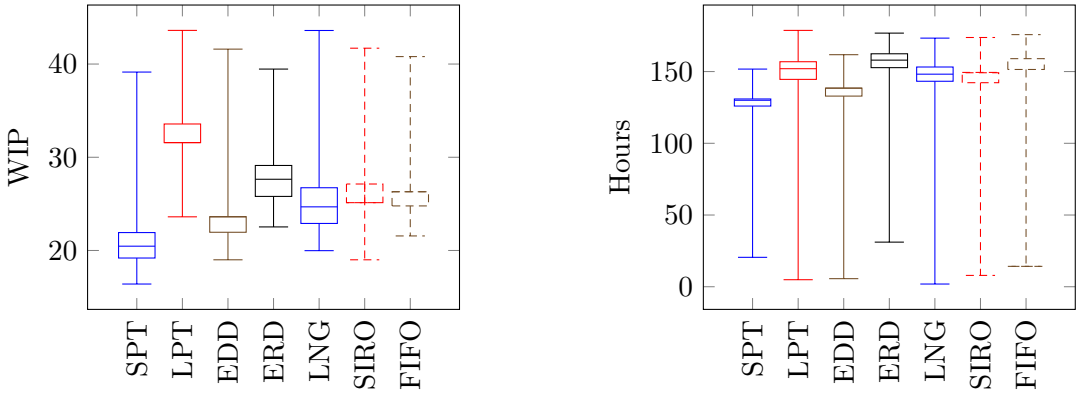
8.3.3 Experimentation results

The results of the experimentation on the rescheduling profiles are given in the box plot Figures of 8.2 and 8.3 below (first and third quartiles were left at the Simio® default values of 25% and 75% respectively.).

When looking at Figure 8.3 (a), it is evident that work-in-process is high (relative to the other performance measures). This due to operators processing tasks with the longest processing time first, many of these tasks arrive at a 'movement assembly' operation which forms a bottleneck when large quantities of gauges arrive simultaneously as only one operator is trained to perform the 'movement assembly' operation. As a consequence throughput for LPT has been reduced (Figure 8.3 (c)). In Figure 8.3 (b) the lower whiskers can be seen reaching a minimum of 1.89. This is due to the fact that the model is initialised with WIP, many near completion, and that the model is forward

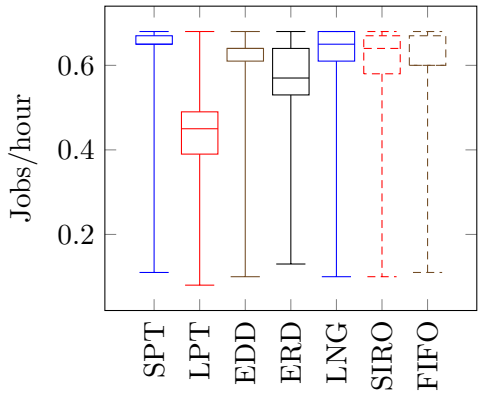
facing (i.e. does not account for the state of the system before the rescheduling point).

With regard to the KN procedure, the dispatching rule of 'shortest processing time' (SPT) consistently outperformed the other dispatching rules and was selected as the 'best' scenario in all but one case. That is, experimentation on RP_3 data resulted in the dispatching rule earliest due date (EDD) being selected when the performance measure was set as 'WIP' with an indifference zone (δ) set to '1'. It must be mentioned however that the EDD rule was not statistically significantly different to the SPT rule.

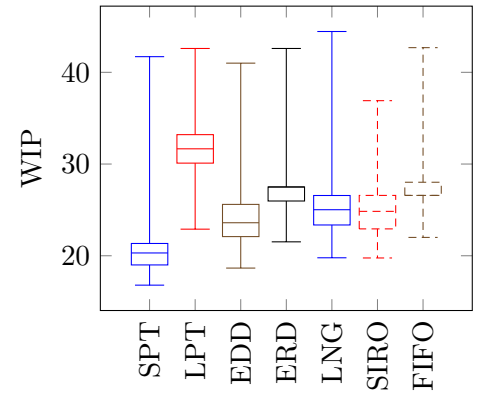


(a) RP_1: WIP

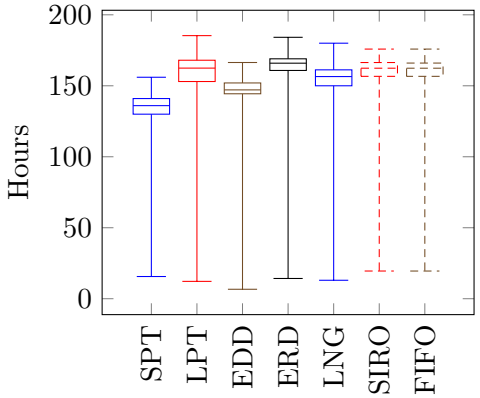
(b) RP_1: Flow Time



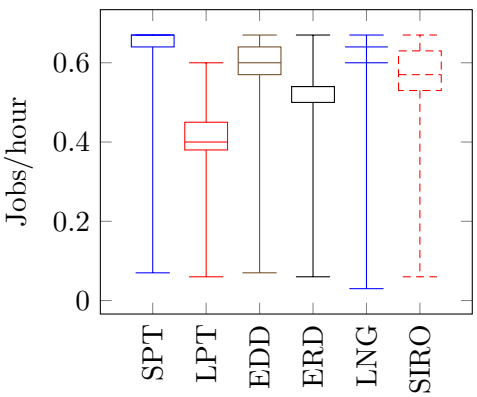
(c) RP_1: Throughput



(d) RP_2: WIP

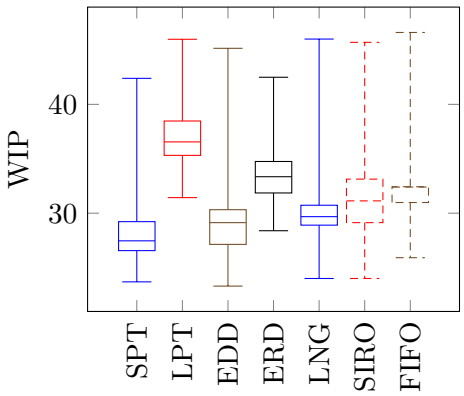


(e) RP_2: Flow time

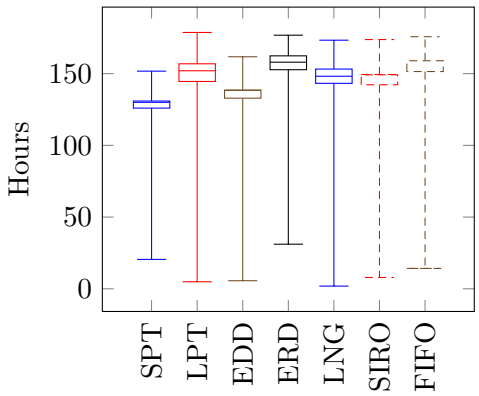


(f) RP_2: Throughput

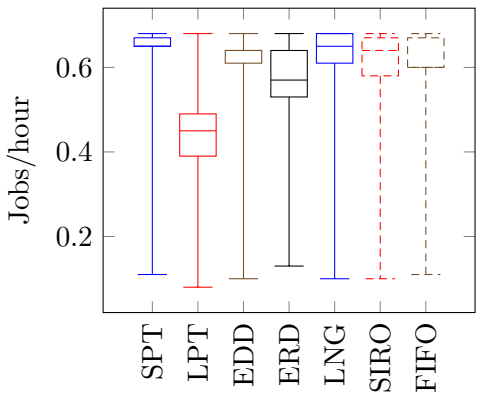
Figure 8.2: Box plots showing the distribution of job expected completion times. Completion times are calculated from the rescheduling instant and include off-shift hours.



(a) RP_3: WIP



(b) RP_3: Flow Time



(c) RP_3: Throughput

Figure 8.3: Box plots showing the distribution of job expected completion times. Completion times are calculated from the rescheduling instant and include off-shift hours.

8.3.4 Priority updates

As discussed in **Chapter 6**, a requirement for the decision support platform was to display the estimated completion times of the jobs in the system. In addition, it was a requirement that floor managers be capable of expediting orders from the mobile client. To demonstrate the effect of expediting an order an experiment was constructed in which job assembly priorities were altered and their completion times then observed.

RP_1 rescheduling data (discussed above) was used for job release and WIP initialisation. From the set of jobs, six were selected at random as candidates for order expediting. In the first scenario all jobs (and their subtasks) were set up with an unmodified priority. Under these circumstances, workstation operators acted in a standard manner by selecting tasks according to their assigned dynamic rank. The results of this scenario, by way of the distribution of each job's estimated completion time, are given in Figure 8.4.

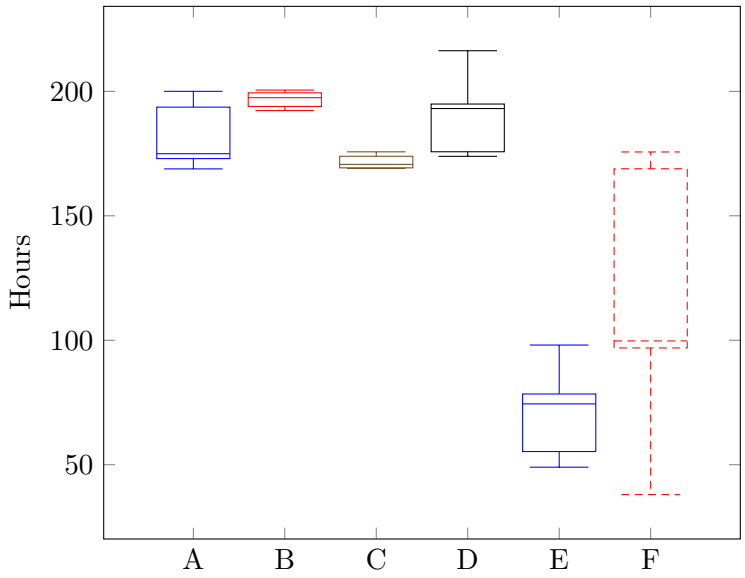


Figure 8.4: Box plots representing job completion times (in hours from the time instant of the rescheduling point and including off-shift intervals). The six jobs are labelled A through F.

In the six remaining scenarios the completion times of the candidate jobs were again monitored, though in each scenario a single job was expedited, i.e., its dynamic rank was overruled. In this case, workstation operators consistently selected this prioritised job first before attending to other jobs in the input buffer. The results of these scenarios are show in box plot format in Figure 8.5. The effect of the prioritisation is seen by

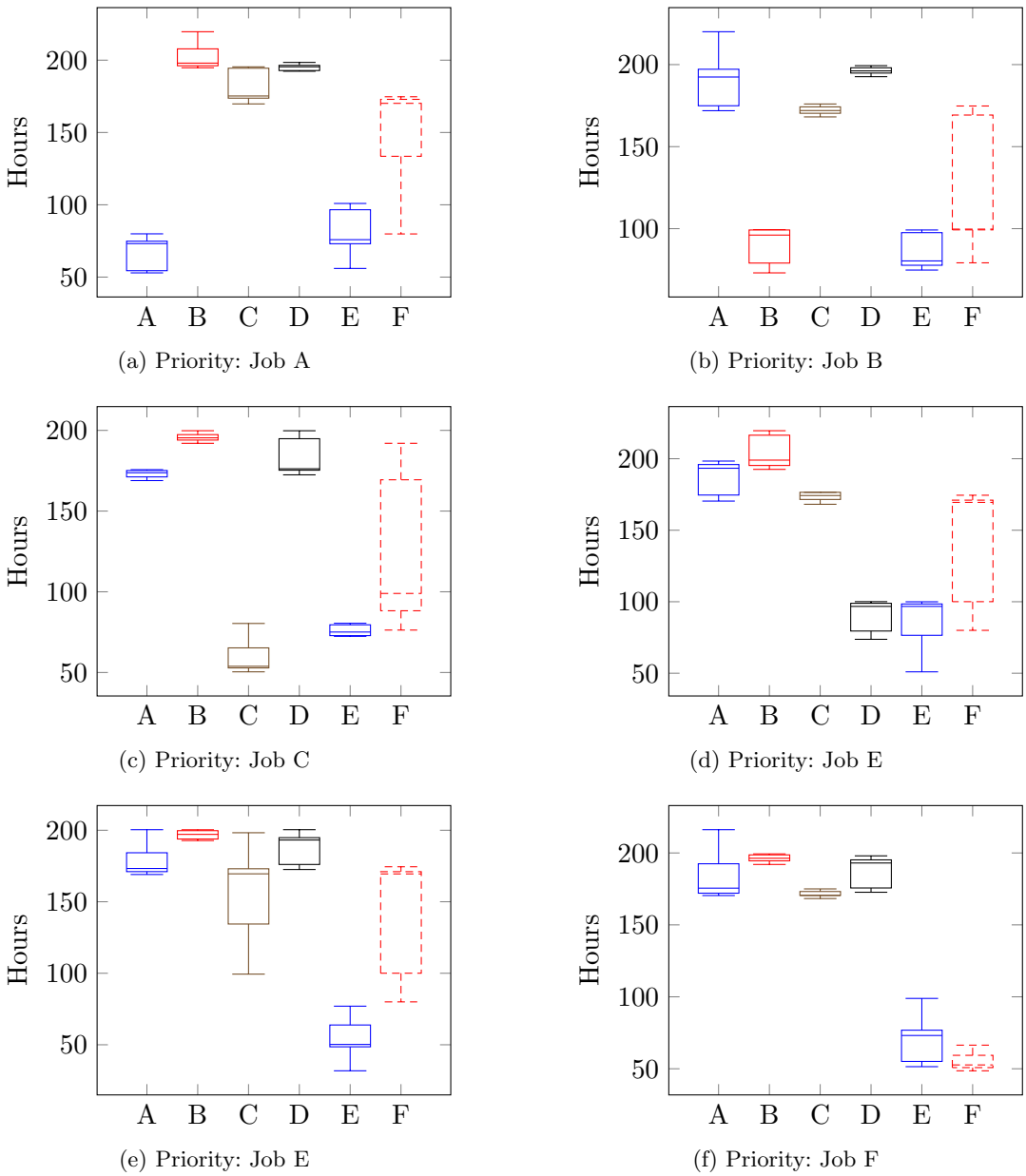
comparing a job's completion time in Figure 8.4, e.g. of job A ($\mu = 179.1$), with its prioritised counterpart, see Figure 8.5 (a) where ($\mu = 66.2$).

Furthermore, verifying, though not proving, that the effect of job prioritisation is statically significant may be performed by comparing the overlap of the confidence intervals of the respective *medians*. The 95% confidence interval (or notch bounds) are defined as (Chambers *et al.*, 1983):

$$M \pm 1.58 \times \frac{\text{IQR}}{\sqrt{n}} \tag{8.1}$$

where M is the median of the performance measure (i.e. derived from replication outputs within the simulation scenario), IRQ is the inter quartile range, and n the number of replications. If the two intervals do not overlap, as is the case between each job and the control scenario, there is strong evidence that the two medians differ at the 95% confidence interval (Chambers *et al.*, 1983).

It is with this that confidence can be had in the fact that sound results are being returned by the simulation model to the RSA, enabling better production planning and operational control of the shop floor, and furthermore, provides reliable up-front visibility into the system for decision making.



(a) Priority: Job A

(b) Priority: Job B

(c) Priority: Job C

(d) Priority: Job E

(e) Priority: Job E

(f) Priority: Job F

Figure 8.5: Box plots showing the distribution of job expected completion times. Completion times are calculated from the rescheduling initialisation instant and include off-shift hours.

8.4 Concluding remarks on Chapter 8

In this chapter a summary of the software architecture for a simulation-based reactive scheduling system was given. Examples of the architecture's conformance to the characteristics of 'levels of abstraction' and 'separation of concerns' were presented. Furthermore, the architecture's attributes of maintainability, scalability, and loose coupling were discussed.

The design of the simulation experiment was described, the application of the Kim Nelson procedure was demonstrated, and the results of the experimentation with the three rescheduling profiles was outlined. The effectiveness of expediting of an order, and having on-demand access to its new completion time, was shown. It was established that at certain rescheduling points a better dispatching rule may in fact exist and can be returned to the shop floor for execution.

Chapter 9

Conclusions

9.1 Thesis summary

The stated aim of this thesis was to design and develop a practical simulation-based reactive scheduling system (RSS). This was achieved through the construction of six primary components, beginning with a simulation model of a real-world pressure gauge assembly plant. Once built, the model was used as the target for experimentation by a scheduling engine. Six local static dispatching rules were used and their effect on the selected performance measure of the system was compared.

Hosting the simulation model was done using a cloud-based reactive scheduling application (RSA). This application was itself the aggregation of a number of sub-components—the most important being the simulation optimisation engine (SOE) and data manager. A RESTful Web service was used to expose capabilities of the RSA to networked front-end devices of the system, effectively offering the scheduler as a service to the shop floor. The placement of the Web service between the front-end devices and RSA was an architectural design point intended to decouple the two systems. Further, to support the data persistence needs of the system, e.g. managing associations between nodes and workstations, a relational database was integrated into the scheduling application.

Coupling the physical shop floor and the software system was achieved using a sensor network and a mobile client. A set of sensor network nodes were prototyped with the purpose of monitoring internal disturbances, i.e. WIP movement. These nodes, with the assistance of workstation operators and RFID cards, captured WIP transition events. All events were uploaded to the Web service for insertion into the system snapshot.

A mobile client application was developed to fulfill two roles, first, to act as an input device for external disturbances, e.g. expedited orders, and secondly, to provide decision support data, generated by the SOE, to floor managers. The application was deployed to a tablet device. Its small form factor and portability made it an idea candidate for a mobile decision support platform.

The research intent of this thesis was to propose a software architecture for a reactive scheduling system. Arriving at a suitable architecture was achieved by through a number of initiatives. First and foremost was the use of the unified process (a software life-cycle model) to guide the software construction effort. By executing a series of iterative increments through the requirements, analysis and design workflows for each of the system's components, while still periodically returning to the global solution—an industrial engineering trait—a *practical* architecture suitable to the development of a reactive scheduling system was created.

A combination of both conceptual and unified modelling language (UML) diagrams were used as a means of communicating the software design decisions, a summary of which was then presented. This suite of diagrams described the system at different layers of abstraction, and made explicit the boundaries and separation of concerns within the system. Furthermore the architecture's desired qualities of maintainability, scalability, and loose coupling were achieved. To that end, the research intent of proposing a software architecture has been met.

It is hoped that the proposed architecture will assist researches and practitioners in creating good architectures for future reactive scheduling systems.

9.2 Future research

The research in this thesis provides platform onto which many improvements can be made. A few suggestions are as follows:

1. The implementation of global dynamic dispatching rules (Section 3.2.1) should be explored as an alternative to the local static rules used in this thesis. Global rules have shown improved performance when compared to localised rule such as SPT (Ouelhadj & Petrovic, 2009).

2. The inclusion of an auto-validation component (Section 2.3.3) into the control loop is essential for real-world deployment (Banks, 1998). This component triggers

the simulation engine to generate a new control law if a deviation (error) between the estimated and realised performance of the system arises. The upper and lower limits of this error term should be investigated.

3. Experimentation could be taken further to investigate the affect of varying the rescheduling window and incorporating different rescheduling policies (Sabuncuoglu & Bayz, 2000; Sabuncuoglu & Kizilisik, 2003).

4. Better single objectives should be investigated, experimental results from Jensen (2001) showed that robust schedules, i.e., a metric which measures of both makespan and stability (starting time deviation and sequence deviations) significantly outperformed schedules based on makespan alone.

5. The research could be taken further by applying using multi-objective optimisation to calculate a Pareto-optimal schedule.

6. Node data buffering. If the network connection is lost during a WIP transaction upload the event is lost. Nodes should persist events locally until a network connection is reestablished .

7. To improve the interaction between mobile clients and the backend system a number of minor improvement could be made, including ETag headers, UUIDS for concurrent conflict resolution, server and client side caching to reduce unnecessary network calls.

References

Baker, N. (2005). Zigbee and bluetooth: Strengths and weaknesses for industrial applications. *Computing and Control Engineering*, **16**, 20–25. 63

Banks, J. (1998). *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. A Wiley-Interscience publication, Wiley. 12, 13, 30, 31, 36, 41, 121, 143

Banks, J., Carson, J.S., Nelson, B.L. & Nicol, D.M. (2004). *Discrete-Event System Simulation*. Prentice hall, 4th edn. 20, 21, 22, 23, 35

Basnet, C. (2011). Selection of dispatching rules in simulation-based scheduling of flexible manufacturing. *International Journal of Simulation Systems, Science & Technology*, **10**, 40–48. 23

Bass, L., Clements, P. & Kazman, R. (2012). *Software Architecture in Practice*. SEI Series in Software Engineering, Pearson Education. 54

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 48

Błażewicz, J., Ecker, K.H. & Pesch, E. (2007). *Handbook on Scheduling: from Theory to Applications*. Springer. 10, 23, 39

Boehm, B.W. (1988). A spiral model of software development and enhancement. *Computer*, **21**, 61–72. 47, 51

Booch, G. (2007). *Object-oriented Analysis and Design with Applications*. Object Technology Series, Addison-Wesley. 107, 178

Chambers, J., Cleveland, W., Kleiner, B. & Tukey, P. (1983). Graphical methods for data analysis. wadsworth int'l. *Group, Belmont, CA*. 117

Chappell, D. (2004). *Enterprise Service Bus: Theory in Practice*. Theory in practice, O'Reilly Media. 96

Chongwatpol, J. & Sharda, R. (2013). RFID-enabled track and traceability in job-shop scheduling environment. *European Journal of Operational Research*, **227**, 453 – 463. 11, 61

Crockford, D. (2006). The application/json media type for javascript object notation (json) request for comments 4627. http://www.ietf.org/rfc/rfc4627.txt. 82

Dashofy, E.M., Hoek, A.V.d. & Taylor, R.N. (2001). A highly-extensible, xml-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, 103–, IEEE Computer Society, Washington, DC, USA. 55

Erl, T. (2008). *SOA: Principles of Service Design*. Prentice-Hall service-oriented computing series, Prentice Hall. 88, 89

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Leach, P. & Berners-Lee, T. (1999). Hyptertext transfer protocol–http/1.1 internet draft standard request for comments 2616. http://www.w3.org/Protocols/rfc2616/rfc2616.txt. 83, 98

Fielding, R., Reschke, J., Lafon, Y. & Nottingham, M. (2014). Hyptertext transfer protocol–http/1.1 internet draft standard request for comments 723x. http://tools.ietf.org/html/rfc7230. 98

Fielding, R.T. (2000). *Architectural styles and the design of network-based software architectures*. Ph.D. thesis, University of California, Urvine. 97, 98

Fowler, M. (2003). *Patterns of Enterprise Application Architecture*. A Martin Fowler signature book, Addison-Wesley. 77, 97

Framinan, J., Leisten, R. & García, R. (2014). *Manufacturing Scheduling Systems: An Integrated View on Models, Methods and Tools*. SpringerLink : Bücher, Springer London. 11, 24

French, S. (1982). *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*. Ellis Horwood series in mathematics and its applications, E. Horwood. 23

Gottschalk, K., Graham, S., Kreger, H. & Snell, J. (2002). Introduction to web services architecture. *IBM systems Journal*, **41**, 170–177. 96, 97

Groover, M.P. (2008). *Automation, Production Systems, and Computer Integrated Manufacturing*. Prentice Hall, 3rd edn. 28, 62

Guo, Z., Ngai, E., Yang, C. & Liang, X. (2014). An RFID-based intelligent decision support system architecture for production monitoring and scheduling in a distributed manufacturing environment. *International Journal of Production Economics*. 62

Hunt, J. (2003). *Guide to the Unified Process featuring UML, Java and Design Patterns*. Springer Professional Computing, Springer London. 52, 67

IEEE (1999). *Standard Glossary of Software Engineering Terminology 610.12-1990*, vol. 1. IEEE Press. 108

Jacobson, I., Booch, G., Rumbaugh, J., Rumbaugh, J. & Booch, G. (1999). *The Unified Software Development Process*, vol. 1. Addison-wesley Reading. 47, 50

Jalote, P., Palit, A., Kurien, P. & Peethamber, V.T. (2004). Timeboxing: a process model for iterative software development. *J. Syst. Softw.*, **70**, 117–127. 49

Jaoua, A., Gamache, M. & Riopel, D. (2012). Specification of an intelligent simulation-based real time control architecture: Application to truck control system. *Computers in Industry*, **63**, 882 – 894. 6, 8, 11

Jensen, M.T. (2001). Improving robustness and flexibility of tardiness and total flow-time job shops using robustness measures. *Applied Soft Computing*, **1**, 35–52. 122

Karl, H. & Willig, A. (2005). *Protocols and Architectures for Wireless Sensor Networks*. Wiley. 146

Kelton, D.W., Smith, J.S. & Sturrock, D.T. (2010). *Simio and Simulation: Modeling, Analysis, Applications*. Boston : McGraw Hill Learning Solutions. 22

Kendall, K.E. & Kendall, J.E. (2011). *Systems Analysis and Design*. Prentice-Hall, 8th edn. xiv, 48, 49, 55, 56, 57, 178

Kim, M.H. & Kim, Y.D. (1994). Simulation-based real-time scheduling in a flexible manufacturing system. *Journal of manufacturing Systems*, **13**, 85–93. 24, 26, 36

Kim, S.H. & Nelson, B.L. (2001). A fully sequential procedure for indifference-zone selection in simulation. *ACM Trans. Model. Comput. Simul.*, **11**, 251–273. 5, 21, 36, 92, 94, 103, 112, 229

Kopetz, H. (2011). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2nd edn. 7

Krige, D. (2008). *Simulation-based online scheduling of a make-to-order job shop*. Master's thesis, Stellenbosch University. 11

Kruchten, P. (2003). *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edn. 51, 54

Laplante, P. & Ovaska, S. (2012). *Real-Time Systems Design and Analysis: Tools for the Practitioner*. John Wiley & Sons, 4th edn. 3, 7, 12

Law, A. & Kelton, W. (2000). *Simulation Modeling and Analysis*. McGraw-Hill series in industrial engineering and management science, McGraw-Hill, 3rd edn. 20, 42

Lee, J.S., Su, Y.W. & Shen, C.C. (2007). A comparative study of wireless protocols: Bluetooth, uwb, zigbee, and wi-fi. In *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*, 46–51, IEEE. 63

Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D. & Mann, W. (1995). Specification and analysis of system architecture using rapide. *IEEE Trans. Softw. Eng.*, **21**, 336–355. 55

Magee, J., Dulay, N., Eisenbach, S. & Kramer, J. (1995). Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, 137–153, Springer-Verlag, London, UK, UK. 55

ManChon, U., Ho, C., Funk, S. & Rasheed, K. (2011). Gart: A genetic algorithm based real-time system scheduler. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, 886–893. 11

Monch, L. & Zimmermann, J. (2004). Improving the performance of dispatching rules in semiconductor manufacturing by iiterative simulation. In *Simulation Conference, 2004. Proceedings of the 2004 Winter*, vol. 2, 1881–1887 vol.2. 26

OMG (2015). OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.5. 55

Ouelhadj, D. & Petrovic, S. (2009). A survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling*, **12**, 417–431. 10, 121

Pandey, R.K. (2010). Architectural description languages (adls) vs uml: A review. *SIGSOFT Softw. Eng. Notes*, **35**, 1–5. 55

Pautasso, C., Zimmermann, O. & Leymann, F. (2008). RESTful web services vs. big web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, 805–814, ACM. 97

Pinedo, M. (2012). *Scheduling: Theory, Algorithms, and Systems*. Springer. 24, 26, 39

Pop, P., Eles, P. & Peng, Z. (2004). *Analysis and Synthesis of Distributed Real-Time Embedded Systems*. Springer. 10

Royce, W.W. (1970). Managing the development of large software systems: concepts and techniques. *Proc. IEEE WESTCON, Los Angeles*, 1–9, reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, March 1987, pp. 328–338. 46, 47

Sabuncuoglu, I. & Bayz, M. (2000). Analysis of reactive scheduling problems in a job shop environment. *European Journal of Operational Research*, **126**, 567 – 586. 122

Sabuncuoglu, I. & Kizilisik, O.B. (2003). Reactive scheduling in a dynamic and stochastic fms environment. *International Journal of Production Research*, **41**, 4211–4231. 122

Schach, S. (2010). *Object-oriented and Classical Software Engineering*. Connect, learn, succeed, McGraw-Hill. xiv, 46, 47, 48, 50, 51, 53, 57

Schwaber, K. & Beedle, M. (2001). *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edn. 48

Sommerville, I. (2011). *Software Engineering*. International Computer Science Series, Pearson. 46, 47

Suwa, H. & Sandoh, H. (2012). *Online Scheduling in Manufacturing: A Cumulative Delay Approach*. SpringerLink : Bücher, Springer London. 10, 23, 24

van Ommering, R., van der Linden, F., Kramer, J. & Magee, J. (2000). The koala component model for consumer electronics software. *Computer*, **33**, 78–85. 55

Vieira, G., Herrmann, J. & Lin, E. (2003). Rescheduling manufacturing systems: A framework of strategies, policies, and methods. *Journal of Scheduling*, **6**, 39–62. 9

Yang, S.H. (2013). *Wireless Sensor Networks: Principles, Design and Applications*. Signals and Communication Technology, Springer London, Limited. 148

Zhang, H., Jiang, Z. & Guo, C. (2009). Simulation-based optimization of dispatching rules for semiconductor wafer fabrication system scheduling by the response surface methodology. *The International Journal of Advanced Manufacturing Technology*, **41**, 110–121. 11

Appendix A

Supplementary information

Table A.1: Operation processing times per processor.

		Operation				
		O_1	O_2	O_3	O_4	O_5
Processor	p_1	1	1	1	2	1
	p_2	1	3	3	1	1
	p_3	1	1	1	2	3

Table A.2: Operational schedule of the industry partner.

Standard day			Half day		
Start	End	Session	Start	End	Session
07:45	08:00	Meeting	07:30	07:45	Meeting
08:00	10:00	On shift	07:45	13:00	On shift
10:00	10:20	Tea			
10:20	13:00	On shift			
13:00	13:30	Lunch			
13:30	16:00	On shift			

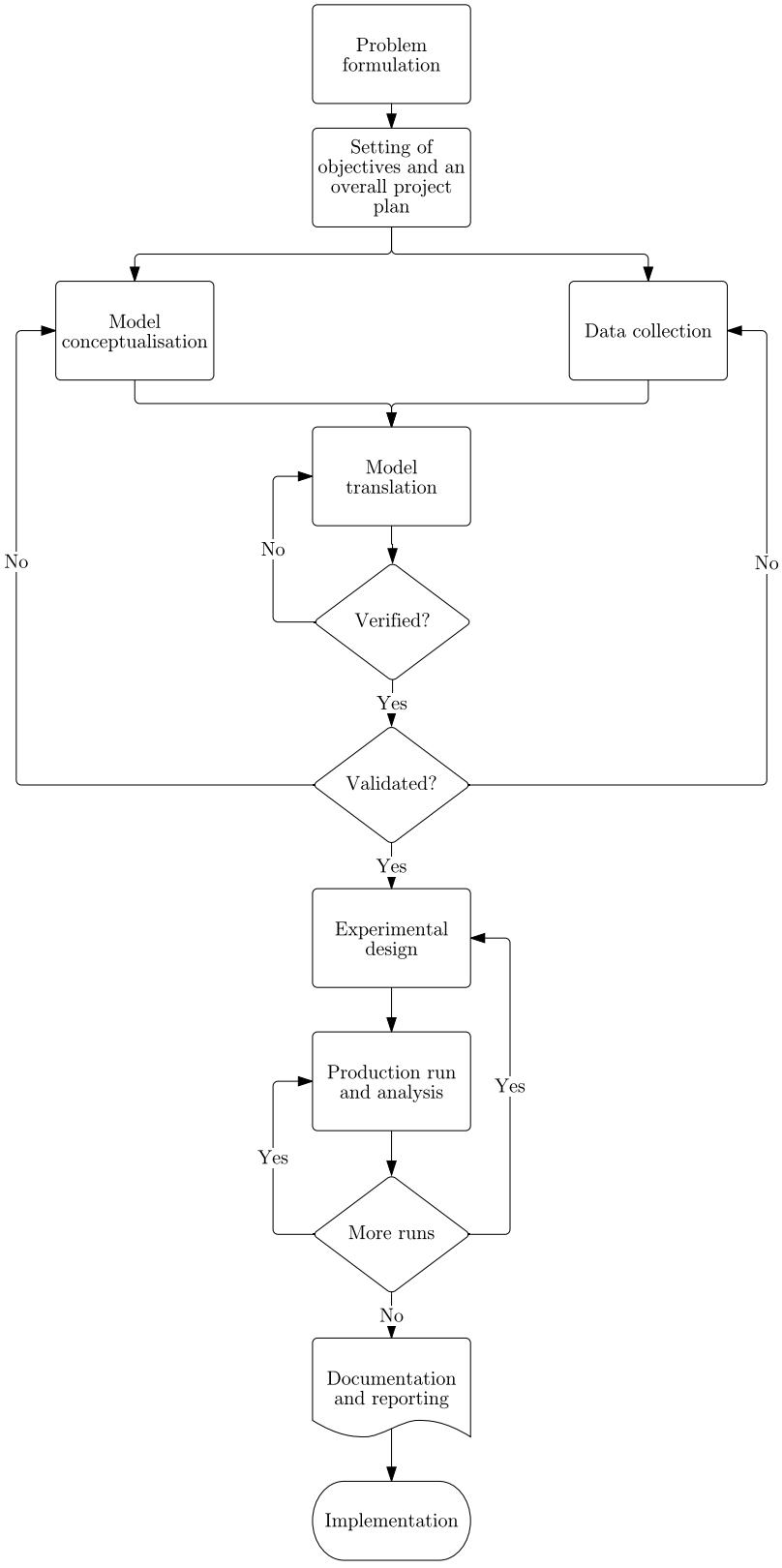


Figure A.1: Steps in a simulation study.

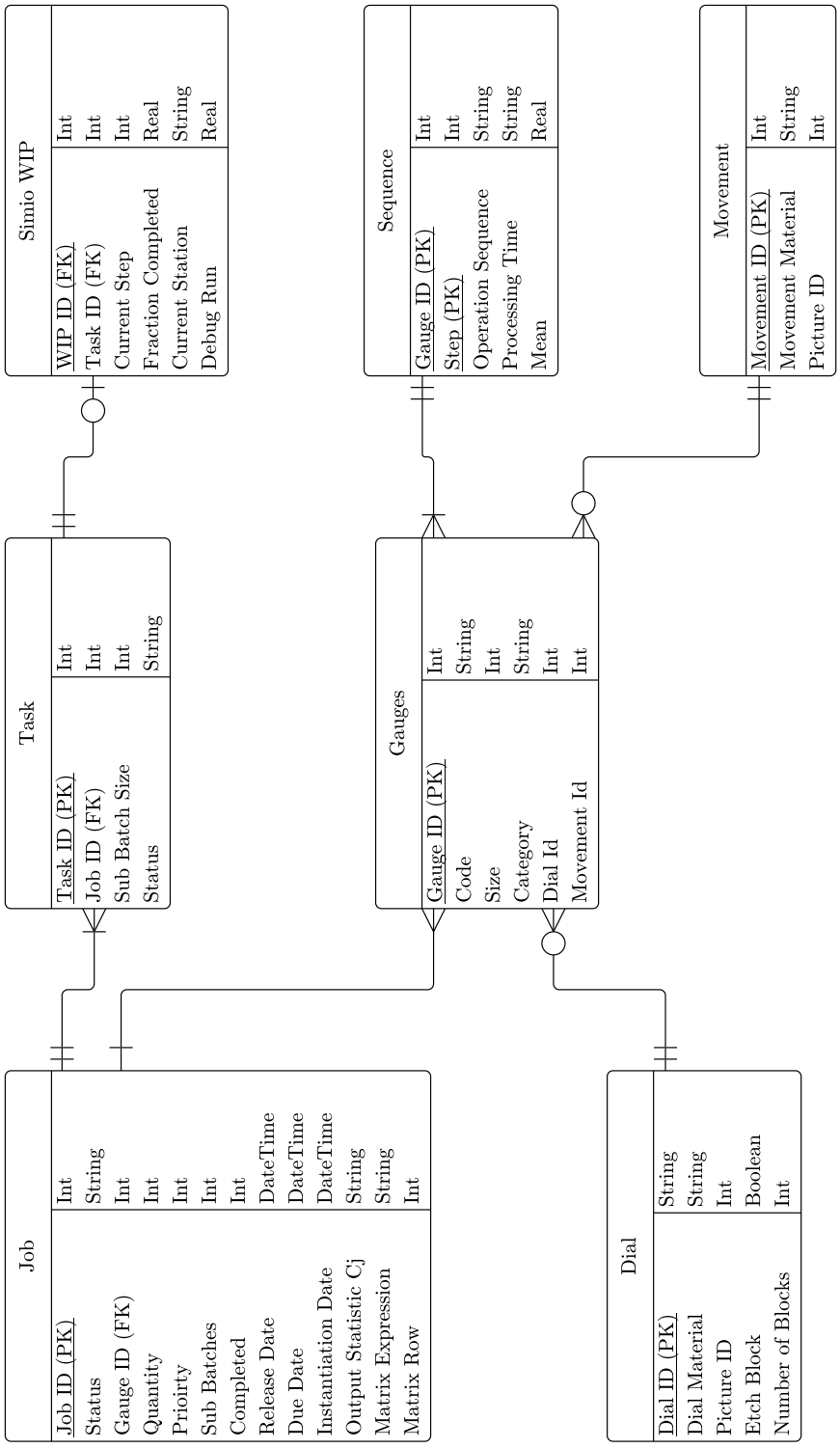


Figure A.2: Entity relationship diagram: Simio relational tables.

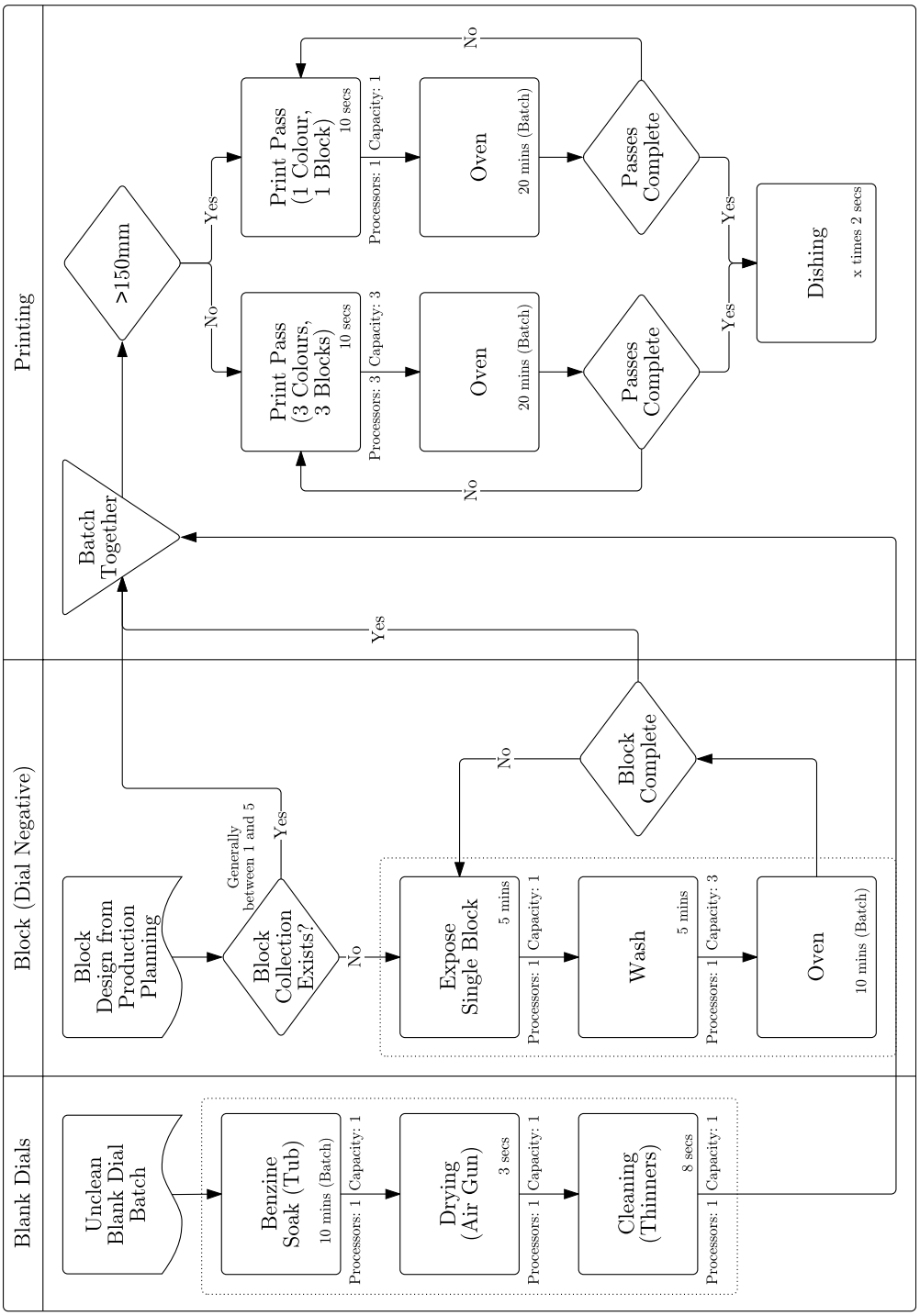


Figure A.3: Process flow diagram: Dial printing process.

Table A.3: Dial face elements

Element	Description	Range	Colour
1	Outer markings	0 psi to 200 psi	black
2	Inner markings	0 bar to 13.79 bar	red
3	Logo	NOSHOK®	black

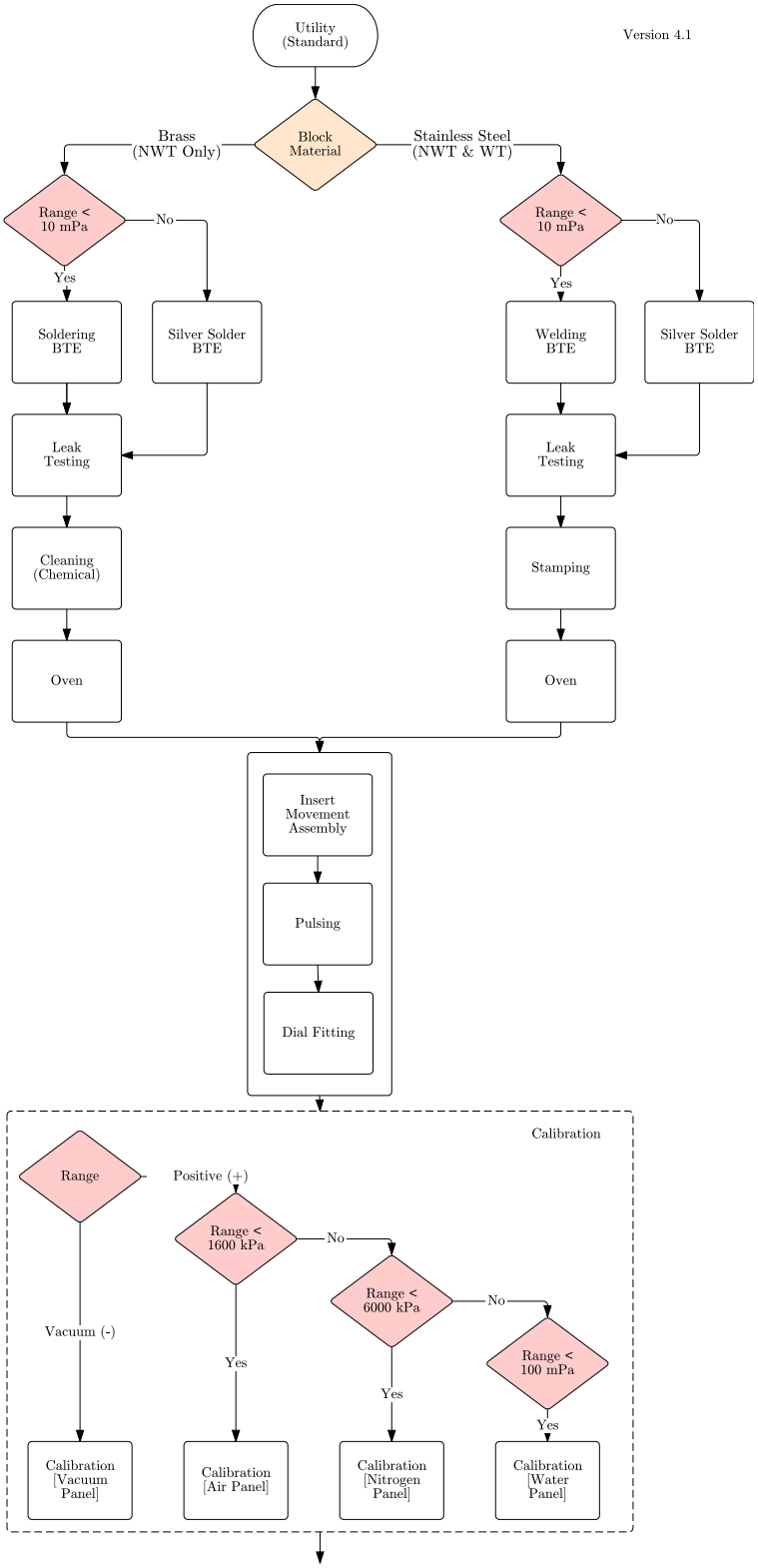


Figure A.4: Process flow diagram: Utility gauge sequence (1/3).

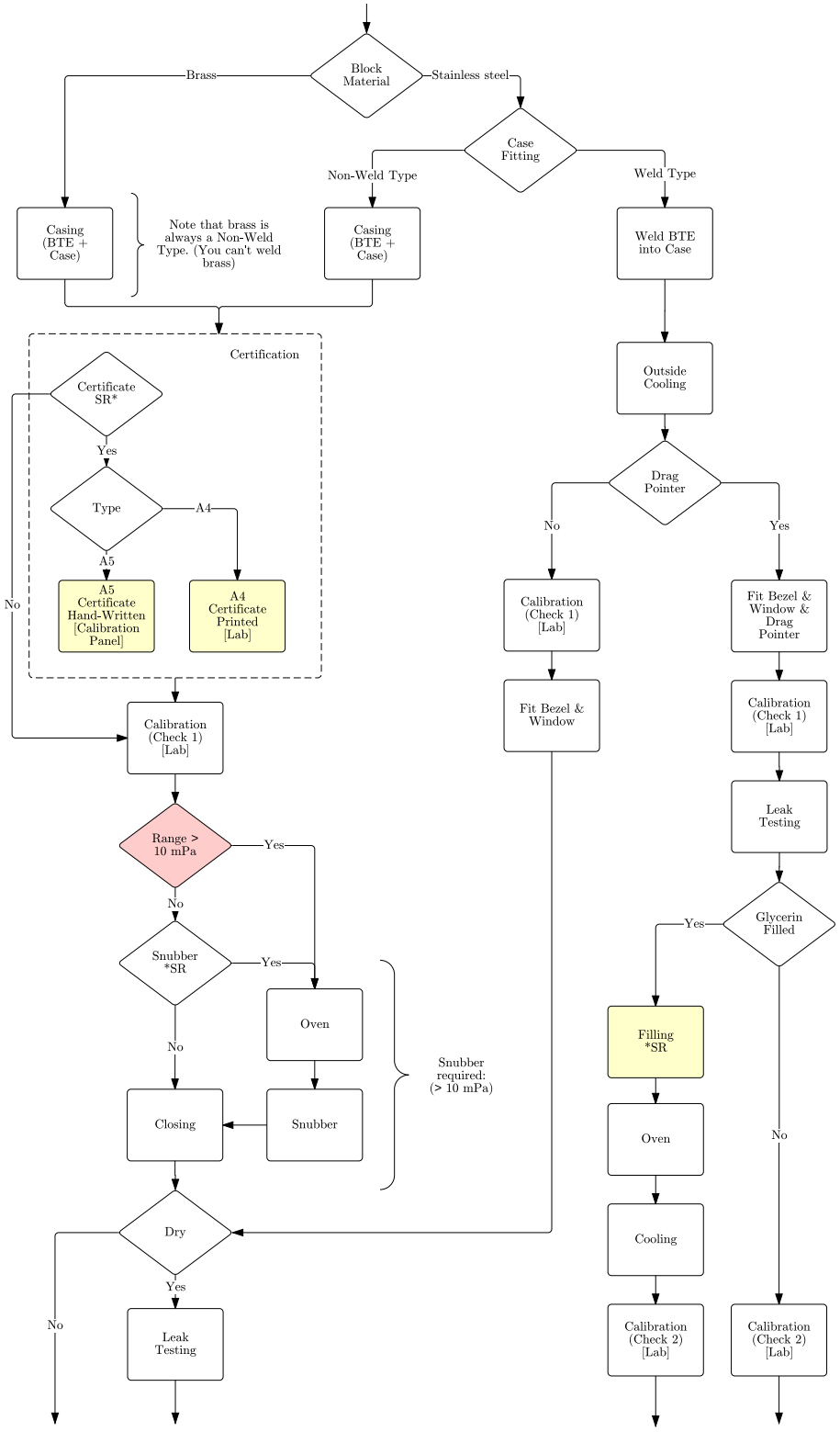


Figure A.5: Process flow diagram: Utility gauge sequence (2/3).

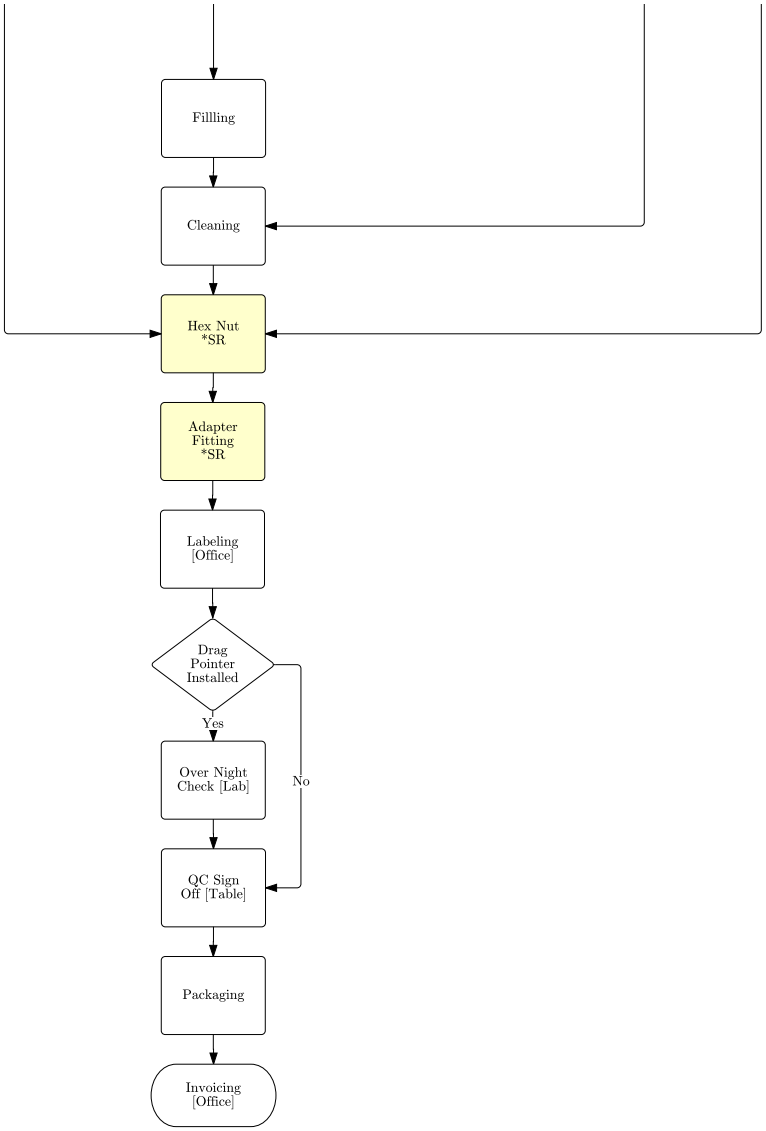


Figure A.6: Process flow diagram: Utility gauge sequence (3/3).

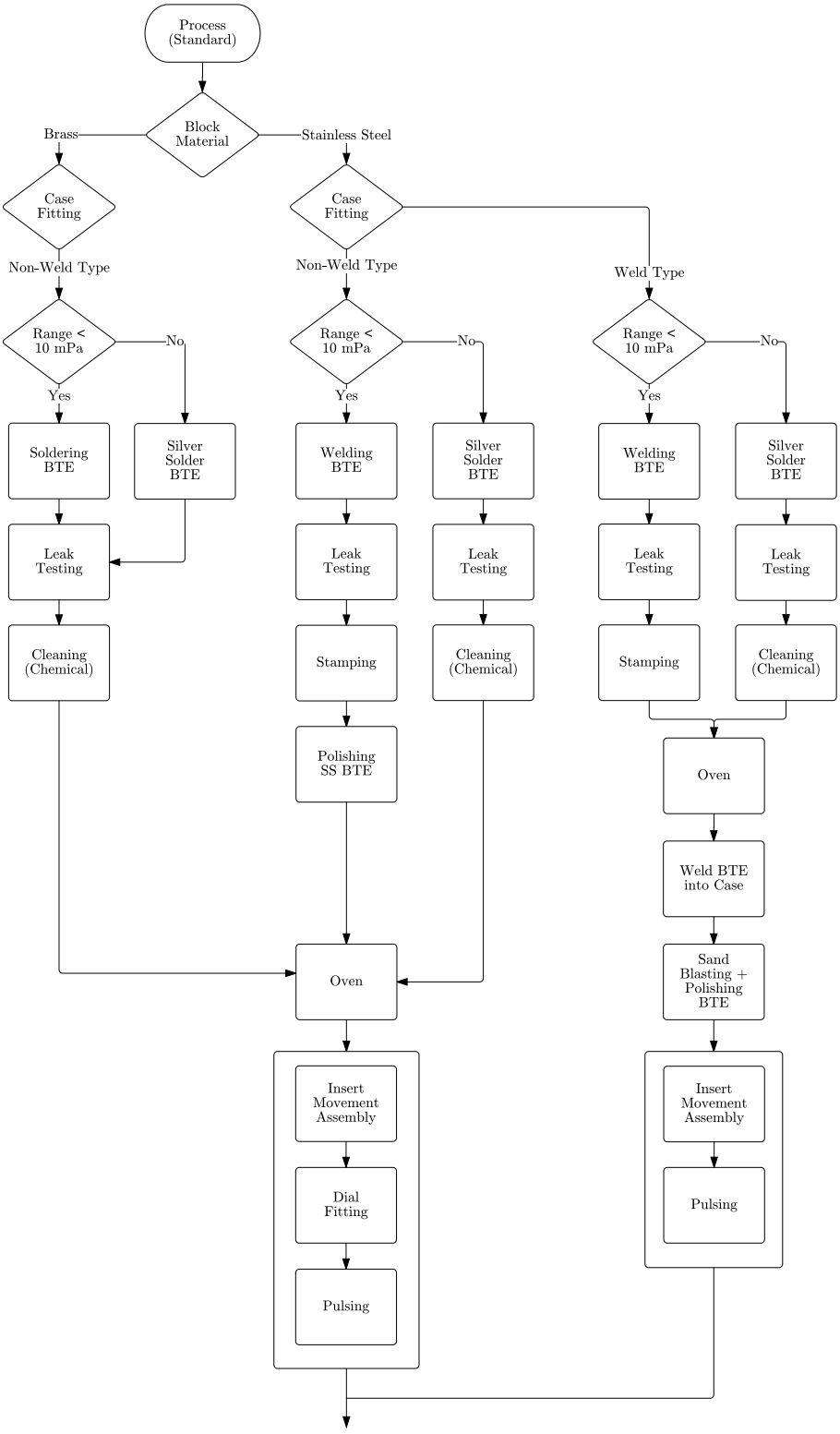


Figure A.7: Process flow diagram: Process gauge sequence (1/3).

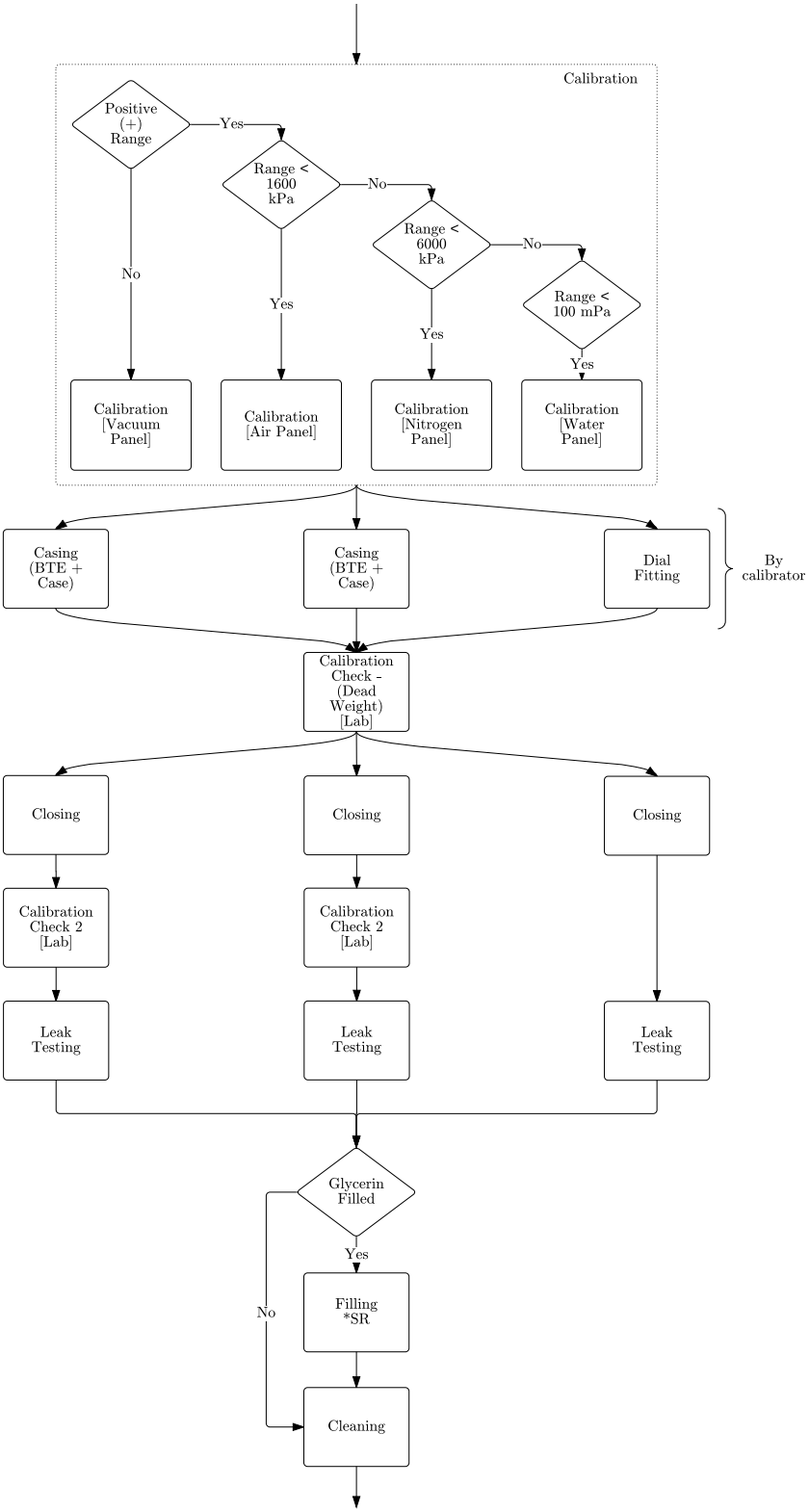


Figure A.8: Process flow diagram: Process gauge sequence (2/3).

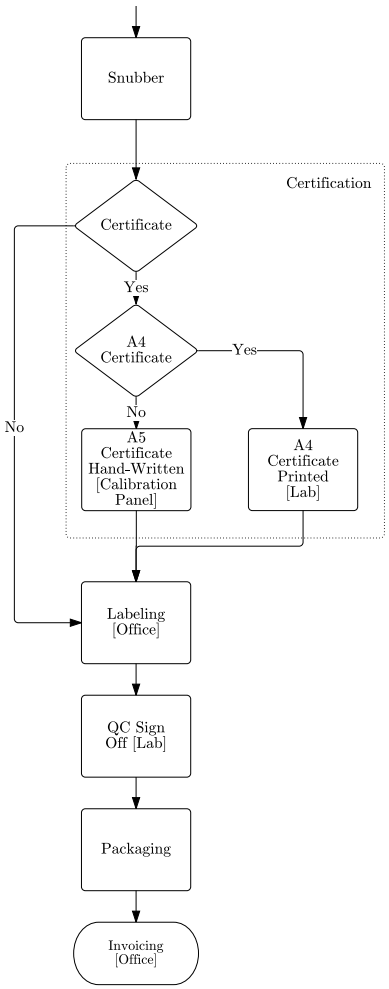


Figure A.9: Process flow diagram: Process gauge sequence (3/3).

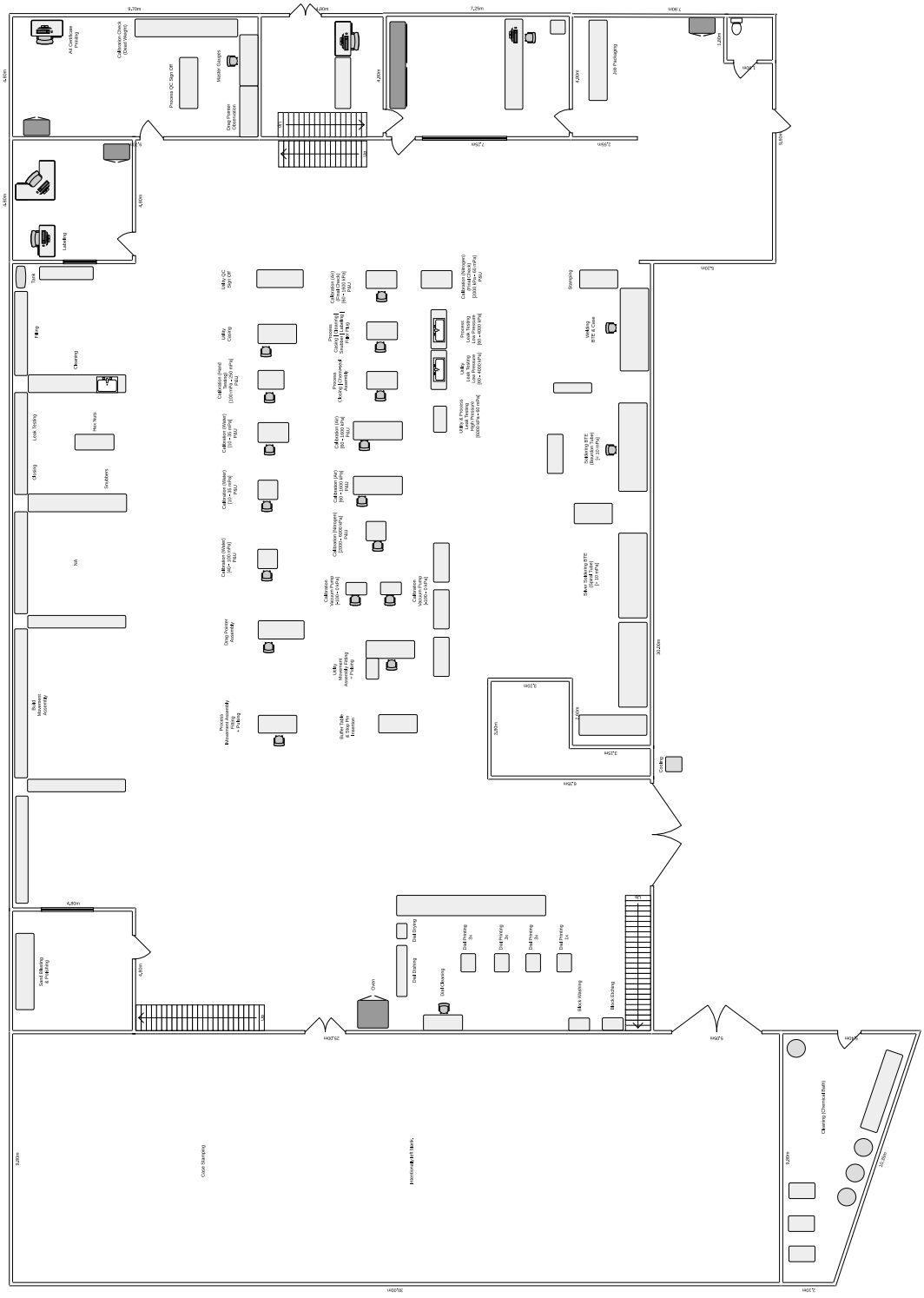


Figure A.10: Floor plan of the pressure gauge assembly facility.

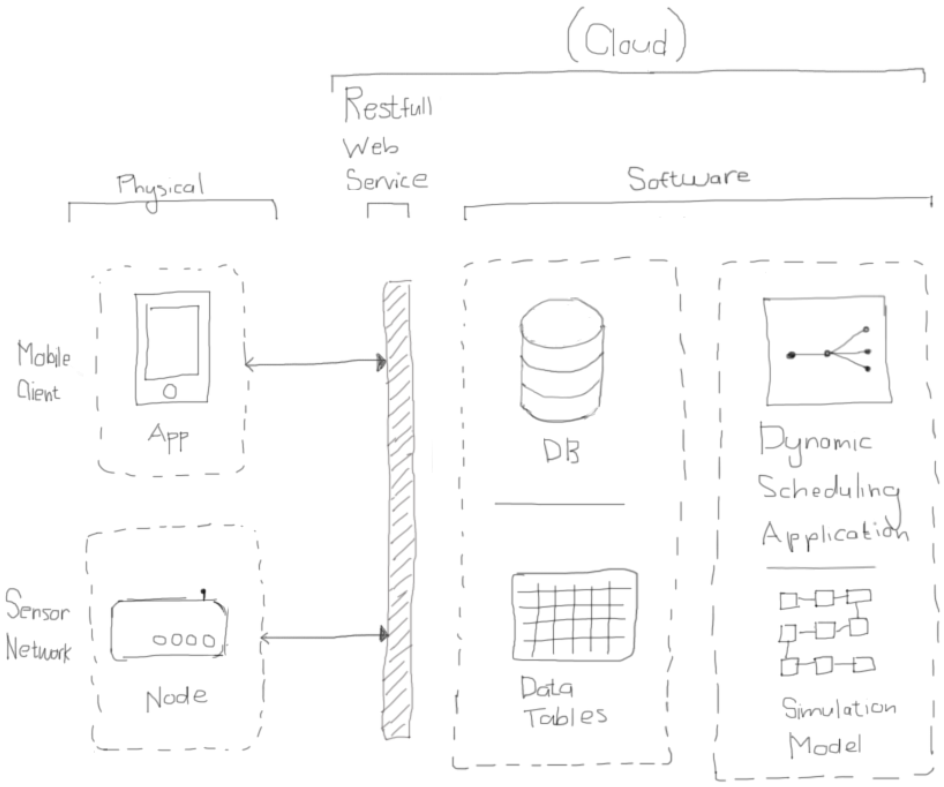


Figure A.11: Hand-drawn conceptual architecture.

Appendix B

Data collection

B.1 Input data

The input data required for the simulation model, as defined by the concept model (Figure 3.5) was obtained by a combination of: conversations with subject-matter experts, observation of the system, and data collection forms. Data relating to the certain system components were collected, namely: a list of candidate operators for each workstation, task operation sequences, task processing time data points, new order information (release date, due date, priority rating, etc.). More detail is given in the list below.

1. **Candidate operators**: A set of candidate operators for each workstation was constructed by interviewing each floor operator. The candidate set includes only the operators that are trained to perform the operations at a workstation. These sets were captured as 'reference lists' in the simulation model.

2. **Operation sequences**: Operation sequences were obtained through interviews with shop floor managers and by manually tracking of tasks through the system. As mentioned in the discussion on the concept model (Section 3.3.4), no process flow diagrams existed at the time of the study, and therefore a period of time was spent creating flow diagrams. This assisting with modelling and understanding the pressure gauge assembly process. Examples of these diagrams are shown in Figures A.7, A.8, and A.9.

3. **Processing times**: After the operational sequences were finalised, data about these sequences needed to be collected. Data collection forms were created with the purpose of gathering timestamp data on the movement of tasks through the system. Operators were asked to fill in a timestamp value at each instant a task was

received at the input buffer, processing *stated* at the processor, and processing was *completed* at the processor. These sheets were attached to tasks and accompanied them through the factory.

Once data collection was complete, analysis of the data was performed. Banks (1998) discussed three techniques for modelling input data:

(a) Assume that the variable is deterministic.

(b) Fit a probability distribution to the data.

(c) Use the empirical distribution of the data.

It was decided that a triangular probability distribution would be sufficient for the purposes of this project. Approximately 25 data points were available (for most of the 38 workstations) from which the triangular distributions could be constructed. For those workstations where data were unavailable, a 'best-guess' distribution was constructed—based on data reverse engineered from the data collection sheets.

4. **WIP snapshots**: To obtain suitable factory state initialisation test data, the state of the factory floor was captured over 5 successive days. Snapshot forms were issued to workstations and operators asked to fill in each form before the day's activities began. This daily snapshot was used to build up a picture of the both the *quantity* and *distribution* of WIP typically found in the system. A summary of the observations is shown in Table B.1.

Table B.1: WIP snapshot

Date	Jobs	Gauges
12 June 2014	53	577
13 June 2014	38	403
17 June 2014	32	405
18 June 2014	26	507
19 June 2014	30	521
Average	34	443

Table B.1 does not indicate the number of new jobs to be released into the system. These were captured by collecting separate production sheets for the for the corresponding day.

5. **Miscellaneous**: Certain happenings in the system were difficult to measure, yet were modelled nonetheless for they influenced the assembly process, for example, floor manager administration. Floor managers were required to attend to administrative tasks throughout day, removing them from the shop floor were they formed part of the assembly operation, such as quality inspection and sign-off. The duration and frequency of these administration tasks could only be estimated by interviewing the floor managers due to the time constrains. Normal distributions were used in such instances.

Input data collection and analysis was difficult in the pressured and sometimes unstructured environment of the industry partner, though much insight into the workings of the assembly line, and the manner in which to approach assembly line operators was learnt during this process.

Appendix C

Sensor network

C.1 Supplementary material

Additional information on the design, analysis, and development of the sensor network.

C.1.1 Operator-node interaction trade-off

An inherent trade-off arises from the required interaction between the operator and the capture device (node). When capturing a WIP state transition (shown in Figure C.1) the operator must interact with the node, temporarily removing the operator from the task at hand, and invariably delaying the task. To establish a baseline for the duration of the transaction capture process a simple experiment was run in which a test subject was asked to capture a sequence of transactions at specific randomly spaced time instants using the prototype discussed in Chapter 5. Durations of 2.6 s, 3.6 s, 4.5 s were obtained for the minimum, average, and maximum respectively.

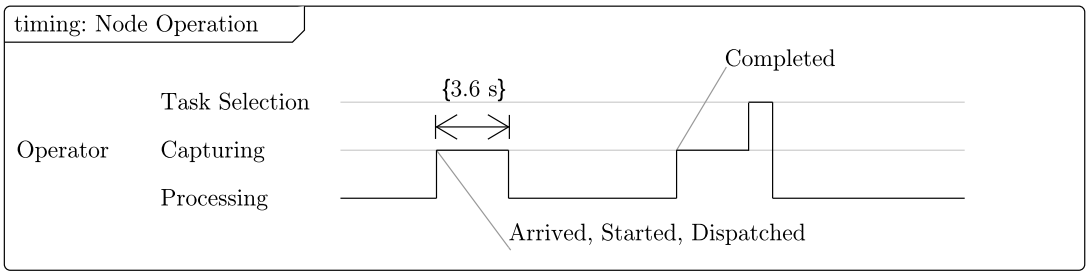


Figure C.1: Timing diagram: Operator-node interaction.

This trade-off between operator interruption and event capture is an important consideration when determining the level of visibility at a workstation. For example, non-critical workstations, i.e., those that do not overly constrain the system, may only need to capture a single state transition, e.g., 'Started'. This may be satisfactory as the objective function of the system may have low sensitivity to the operation performed at that workstation. Conversely, a workstation to which the system exhibits high sensitivity, as in the case of a bottleneck, may demand a higher level of visibility (at the expense of operator interaction). By capturing more state transitions at sensitive workstations the scheduling engine may be able to improve its scheduling capability since the simulation model has a finer level of detail in a key area for model initialisation. This topic is recommended for future research.

C.1.2 Capturing the state transition instant

Attempting to log the state transition instant raised an issue found to be common to all sensor networks, and in fact real-time systems at large. That is, clock synchronisation across distributed devices. A solution needed to be found to ensure all devices in the network subscribed to a common timescale.

- **Attempt 1**: Log the transition instant using the Waspmote's internal real-time-clock (RTC)[1]. Due to the prototyping environment in which the nodes were repeatedly unplugged and reconfigured, resetting global time became time consuming. The time-stamping of events was then deferred to the server.
- **Attempt 2**: Log the transition instant using the timestamp of the instant the event hits the server [2]. This solution was easier to manage, but shifted timestamps backward due to network latency between the sensor network and Web service.

With global network latency averaging 250 ms, the time shift was considered to be negligible.

[1] A minor, but important, issue to take note of is that crystal oscillators of micro-controllers have the tendency to drift. Thus without periodic correction, the cumulative effect may be substantial (between 1 and 100 parts per million which correspond to drift rates of 1 s every 11,6 days or 1 s every 2.78 h respectively (Karl & Willig, 2005)).

[2] A time zone correction was needed as the server was hosted in a data centre in Western Europe.

C.1.3 Displaying the recommended task

The original design of the DSS intended for the node to be the main delivery mechanism for the dispatching rule to the shop floor operators. In the node concept mockup (Figure C.4), an LCD is shown displaying the unique identifier of the task id recommended by the SOE at that point in time. Upon completing the task a hand, the operator would select the corresponding task in the input buffer. This action would have formed the forward link between the SOE and the operator, and allowing the process to be governed.

At the time of writing, a technical issue with the memory buffer of the WiFi module prevented the node from displaying the recommended task, and as a result, the presentation of the recommended task was shifted to the mobile client. In doing so, the concept of presenting the recommended task to an operator could still be demonstrated and the research goal achieved.

C.1.4 Interacting with Waspmotes setup and loop fuctions

Integral to software was the usage of Waspmote's two main functions, `setup` and `loop`. `setup` is called *once* upon device start-up, whereas `loop` is called *repeatedly*. `setup` is responsible for the initialisation of the Waspmote, which involves: defining the input and output pins, informing Waspmote of the type of communication modules that are attached, followed by activating, initialising these modules, and finally discovering the device's unique serial identifier.

After the device has been properly initialised the `loop` function is called by the microcontroller. Logic was implemented within the `loop` function to continually monitor the input pins for user interaction. If a state transition button is pressed, e.g., "Received", `loop` is interrupted, the RFID module is activated, and the transaction capture sequence is initiated.

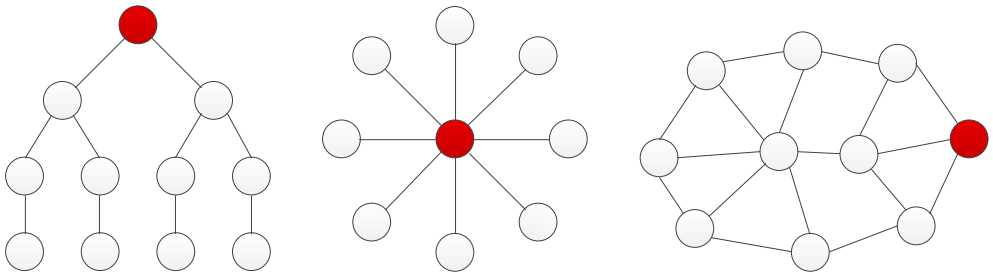
C.1.5 IP address assignment

IP address issuing was handled using dynamic host configuration protocol (DHCP). The primary node was set up to issue IP addresses ranging between 192.168.1.0 and 192.186.1.150. Nodes within range of the access point were dynamically assigned an available IP address, enabling them to communicate over HTTP. WAP security protocol was also implemented.

C.1.6 Wireless network protocols and topologies

Sensor networks, regardless of their topology, must contain a single *primary* node (also referred to as a coordinator) whose purpose is to: form the network, define the communication logic, and secure the network. *End-devices* are devices whose purpose it is to collect data from the environment and relay it back to the primary node. Data is communicated to the external network (most commonly the internet) via an interface with the primary node.

The choice in topology, i.e., logical structure, of a sensor network is constrained by the architecture of the physical layer. IEEE 802.15.4 supports star, tree, cluster tree, and mesh topologies, whereas IEEE 802.11 supports only star and tree (Yang, 2013), see Figure C.2. Since IEEE 802.11 devices are limited to one-to-one communication, i.e., a primary node must talk directly to an end device, the node prototype was constrained to using a star topology as illustrated in Figure C.2 (b).



(a) (b) (c)

Figure C.2: Network topologies: (a) Tree (b) Star (c) Mesh.

An important advantage of the mesh configuration, specifically for manufacturing environments, is that each end-device need not to be in range of the primary node (red)—as in the case of the star topology. End-devices only need be in range of some other end-device which itself forms part of the mesh. This is ability to relay data allows for larger networks and for better positing of the primary node. It is recommended that in future research, IEEE 802.15.4 radios are used in the mesh topology.

C.2 Technical specifications

Table C.1: Technical specifications: Waspmote^{TM} Pro microcontroller.

Property	Description
Microcontroller	ATmega1281
Frequency	14 MHz
SRAM	8 KB
EEPROM	4 KB
FLASH	128 KB
Weight	0.02 kg
Clock	RTC (32 kHz)
Temperature range	[−10 °C, 65 °C]
Interface	UART
Dimensions	73.5 mm × 51 mm × 13 mm
Voltage	3.3 V −4.2 V
Consumption	ON: 15 µA, Sleep: 55 µA

Table C.2: Technical specifications: 13.56 MHz RFID module.

Property	Description
Support	ISO14443A/MIFARE
Frequency	13.56 MHz/NFC
Read/Write distance	5 cm
Max capacity	4 kB
Interface	UART
Tags	Cards, Keyrings, Stickers

Table C.3: Technical specifications: 802.11b/g – 2.4 GHz WiFi module.

Property	Description
Protocols	802.11 b/g – 2.4 GHz
TX Power	0 dBm – 12 dBm
RX Sensitivity	−83 dBm
Antenna connector	RPSMA
Antenna	2 dBi
Security	WEP, WPA, WPA2
Topologies	AP & Adhoc
IP Setup	DHCP, Static

Table C.4: Node prototype: Bill of materials.

Item No.	Description	Quantity
1	Waspmote^{TM} Pro microcontroller	1
2	Libelium® 802.11b/g – 2.4 GHz WiFi board	1
3	Libelium® 13.56 MHz RFID module	1
4	LED red	1
5	LED green	1
6	Push button	5
7	Resistor – 10 kΩ	5
8	Resistor – 1 kΩ	2
9	Connectors cables	9
10	Copper wire 10 cm	1
11	Breadboard (80 mm × 60 mm × 8 mm)	1

C.3 Prototype illustrations

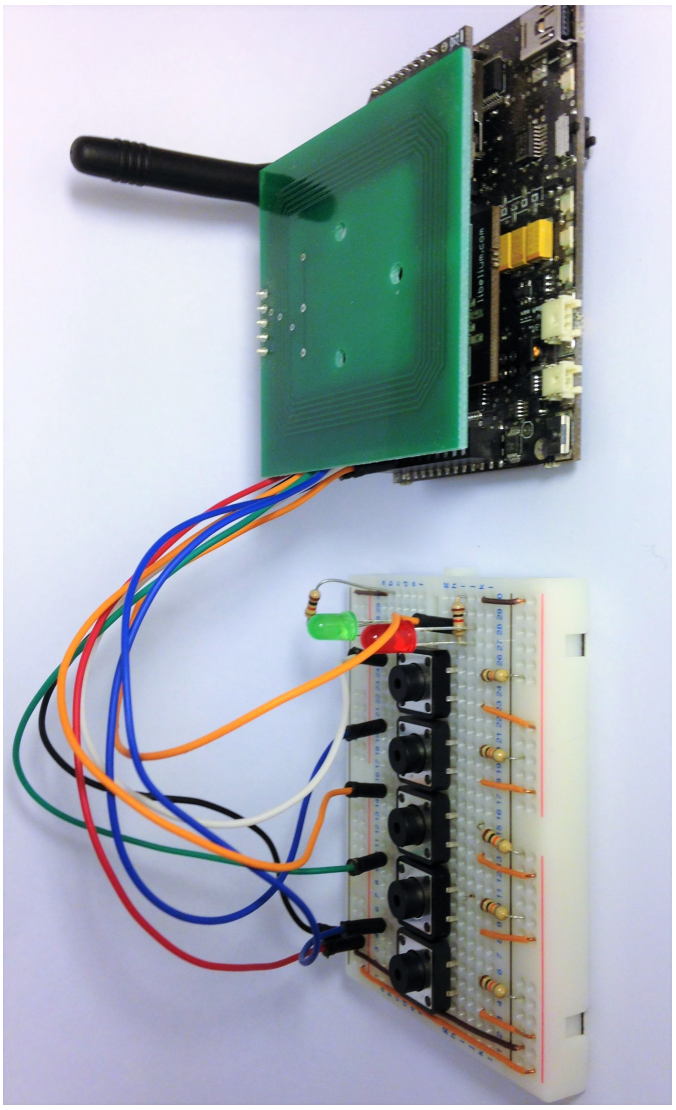


Figure C.3: Photograph of node prototype showing the Waspmote micro-controller, state transition buttons, WiFi antenna (black), and RIFD antenna (green).

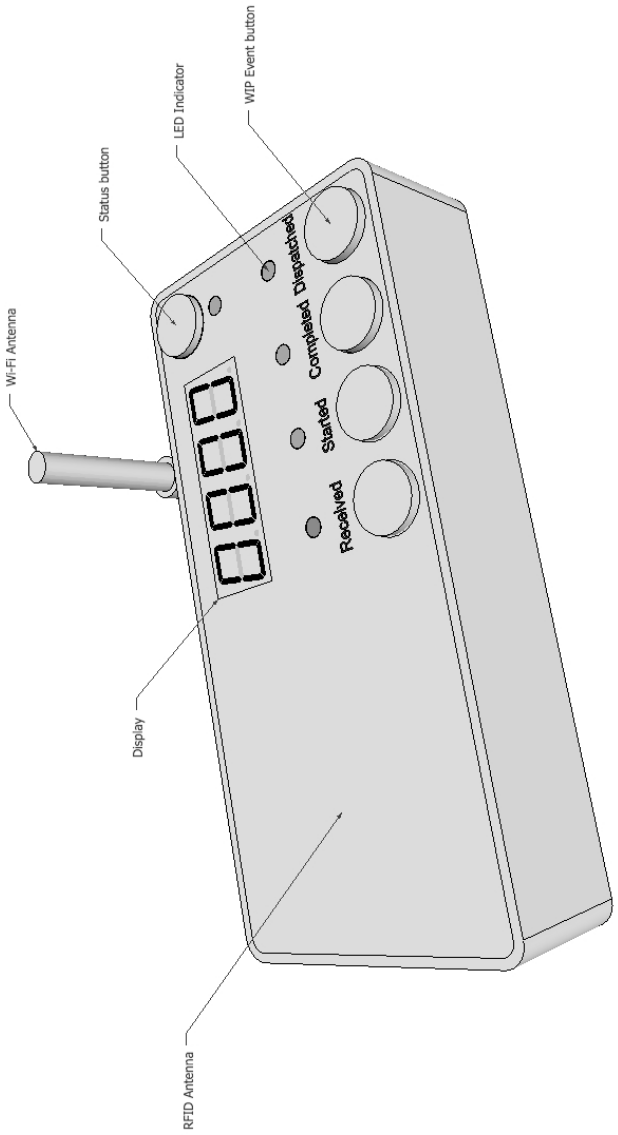


Figure C.4: Node design mock-up. Dimensions are approximately $150\,\text{mm} \times 90\,\text{mm} \times 60\,\text{mm}$ (l x b x h). An LCD display is shown above the state transition buttons. The purpose of the LCD was to present the recommended task to the operator. The feature was not included due to the time constraints.

C.4 Diagrams of the UML modeling process

Table C.5: Logging WIP state transition.

Use case name:	WIP state transition
Unique Id:	N_001
Category:	State capture
Actor(s):	Operator
Level:	Blue
Description:	Allow workstation operator to capture a WIP state transistion.
Triggering event:	A WIP item has transitioned from one state to another (defined by the event resolution).
Trigger type:	Internal, Temporal

Steps Performed (Main Path)	Information Required
1. Operator actions, or notices, a WIP state transition.	State transition category: *Received/ Started/ Completed/ Dispatched*
2. Operator picks up associated RIFD card.	N/A
3. Operator presses the corresponding state transition button capturing the state and activating the RFID antenna (zero to five centimetres).	State transition category. Unique Id of Waspmote.
4. Operator swipes the RFID card over antenna area.	Unique Id of RFID card.
5. Operator awaits confirmation from LED blink pattern.	N/A
6. WIP transition is uploaded to the server.	N/A

Preconditions:	Node is in the neutral state and is not currently capturing a transaction.
Postconditions:	Operator has successfully captured a WIP state transition and the event has been uploaded to the server.
Assumptions:	RFID card is associated with the WIP item. Operator pressed correct state transition button.
Success guarantee:	WIP state transition communicates with Web service.
Minimum guarantee:	LED blink pattern indicating success.
Requirements met:	Captured state transition.
Outstanding issues:	How should rework be handled? Audible success indicator.
Priority:	High
Risk:	Medium

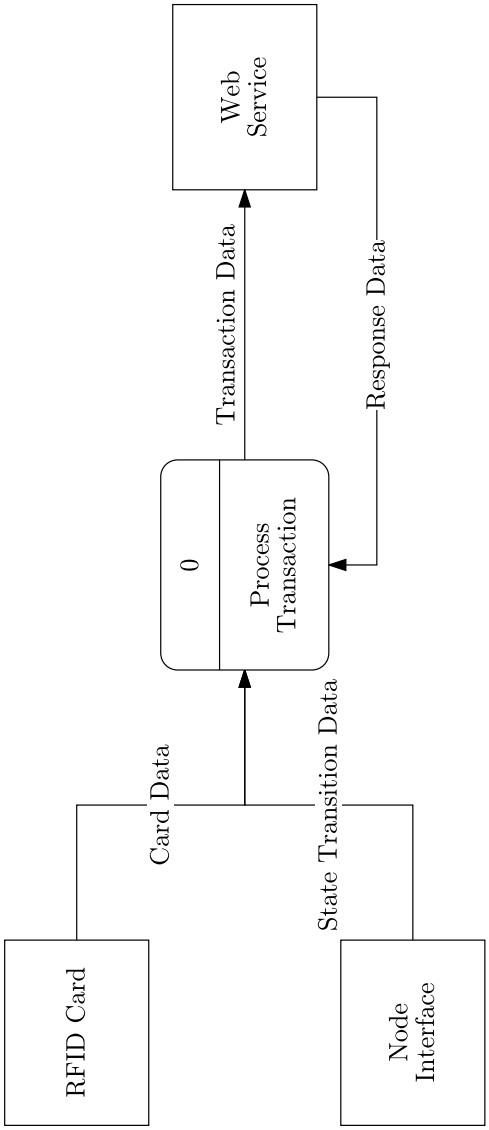


Figure C.5: Data flow diagram: Node — Context.

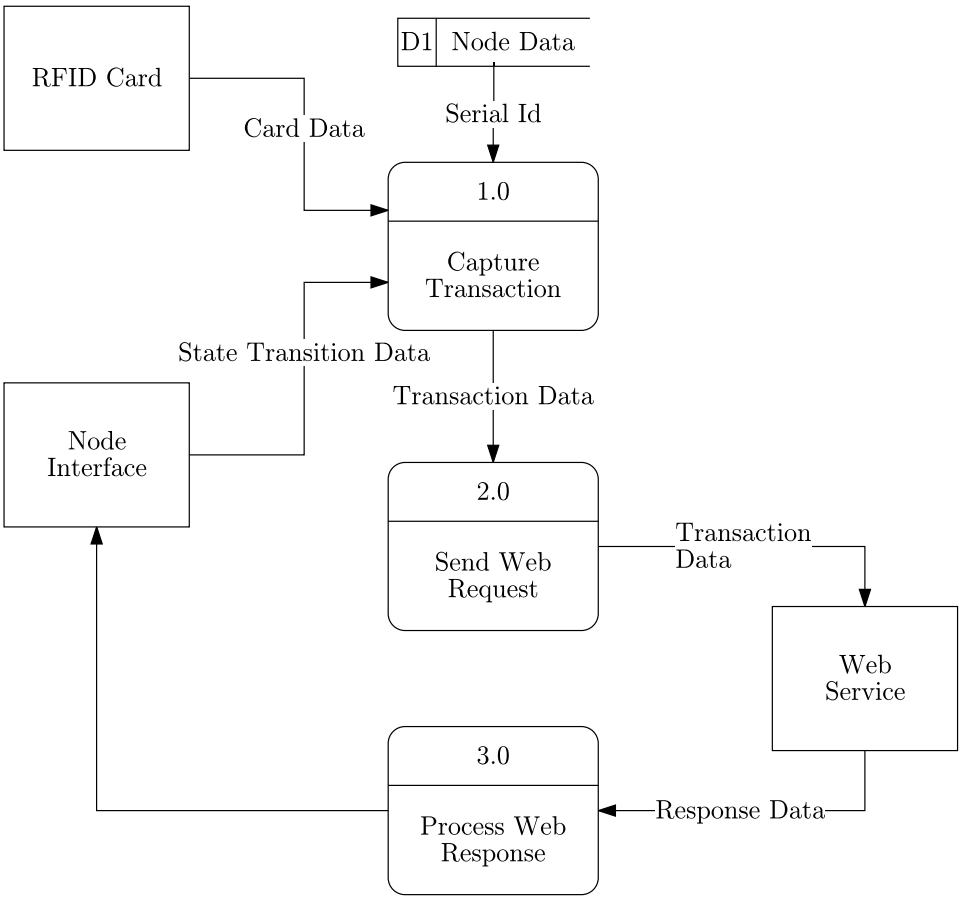


Figure C.6: Data flow diagram: Node — Level zero.

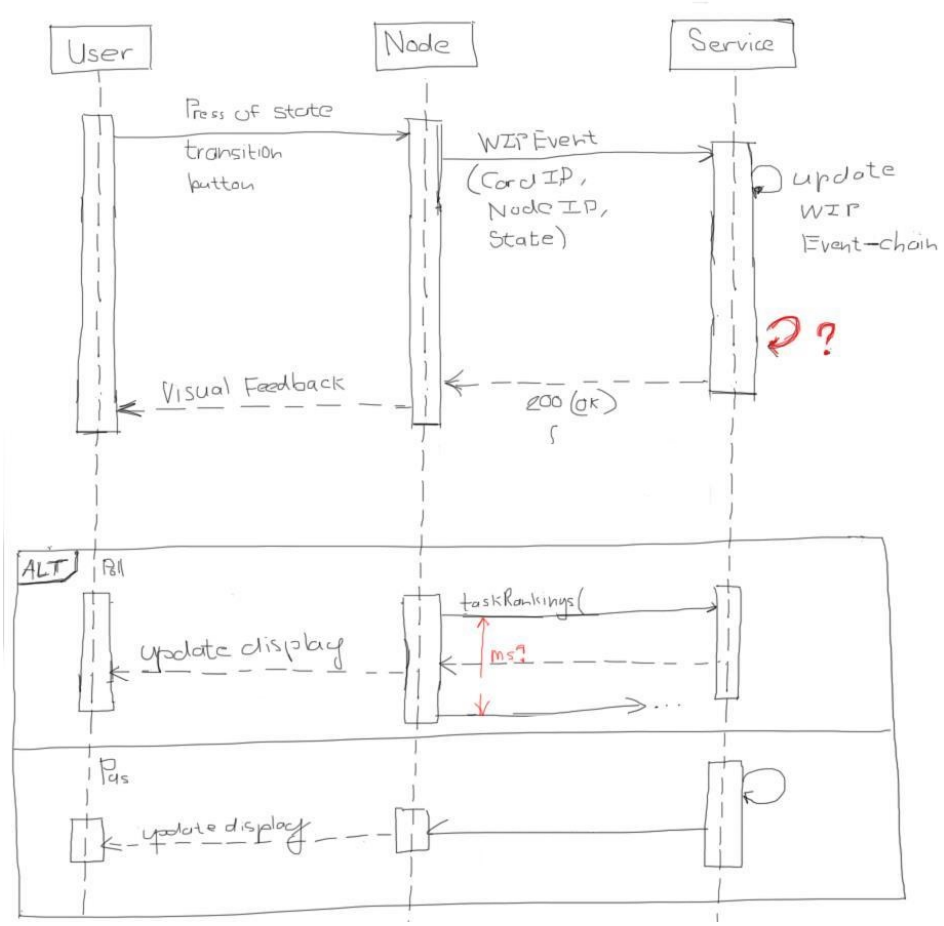


Figure C.7: Sequence diagram: Node — first iteration. In the lower half of the sequence diagram, events are shown for updating the node's display. This feature was descoped from the node due to time constraints. Its inclusion in the analysis and design workflows is intended for future research.

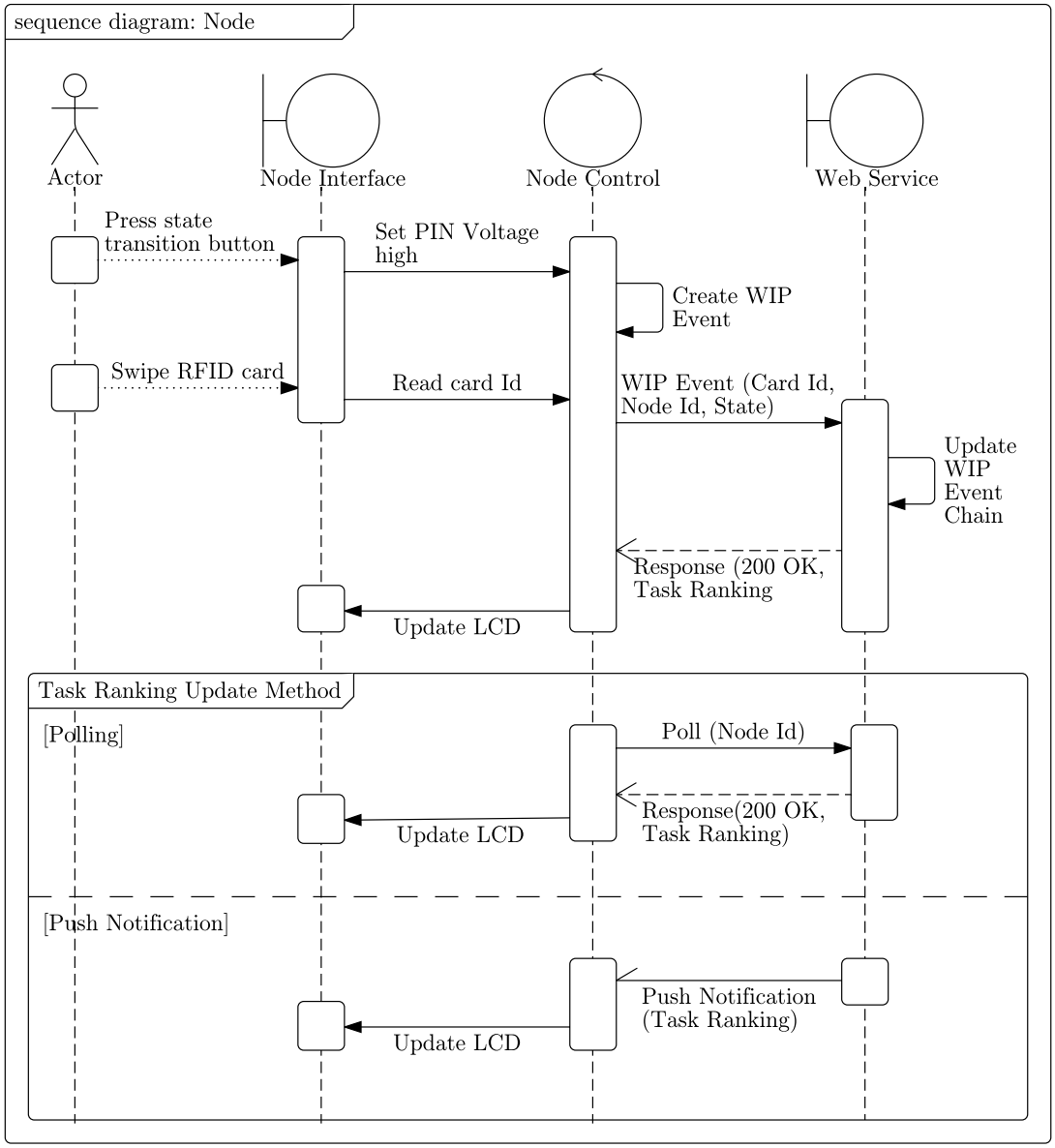


Figure C.8: Sequence diagram: Node — final iteration.

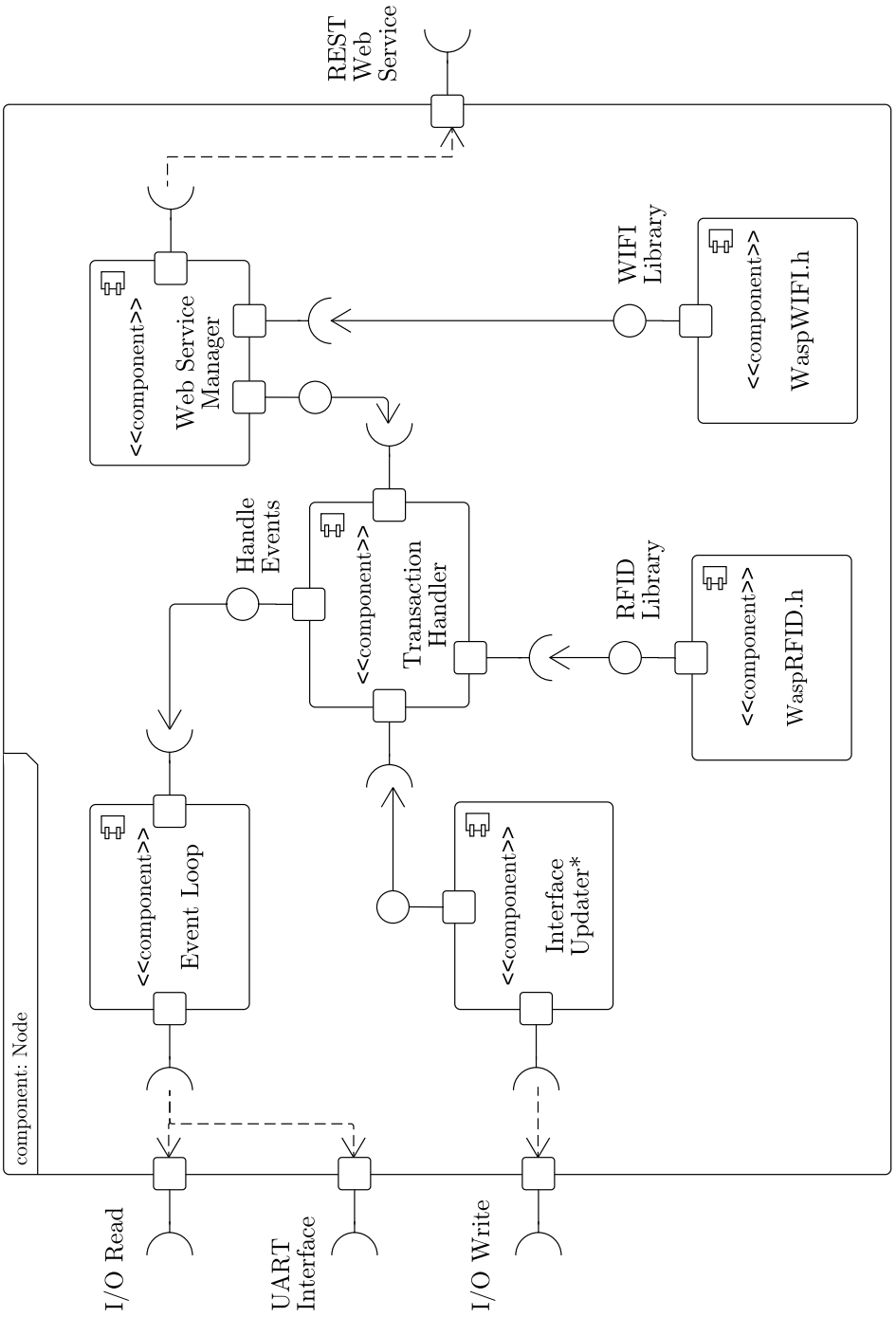


Figure C.9: Component diagram: Node.

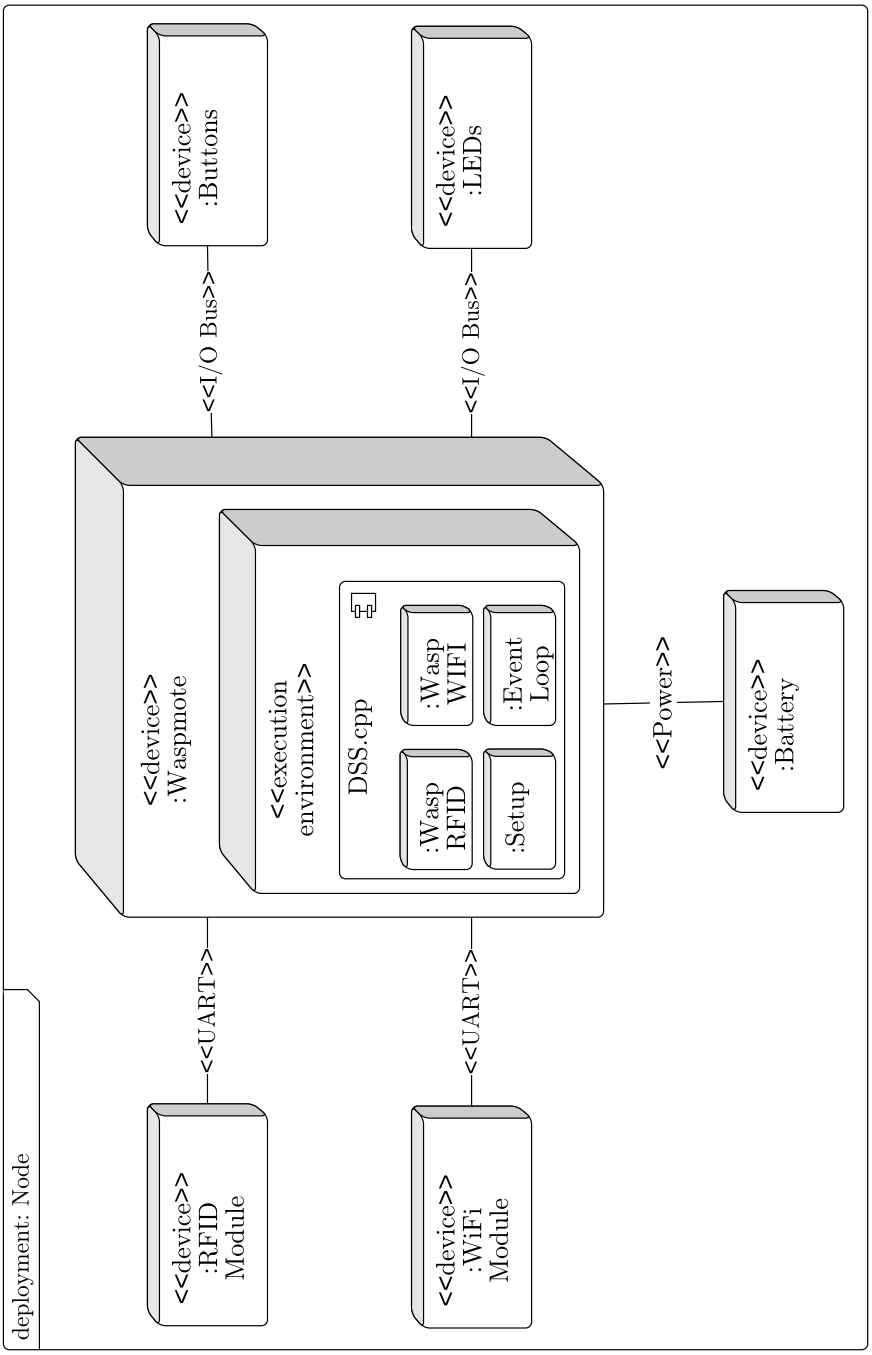


Figure C.10: Deployment diagram: Node.

Appendix D

Mobile client

D.1 Supplementary material

Additional information regarding the mobile client is provided in this appendix.

D.1.1 Summary of the functional requirements

Table D.1 shows a summary of the requirements of the mobile client.

D.1.2 Device selection

Modern smart-phones typically exhibit a diagonal screen size of between 8.89 cm and 12.7 cm (3.5 in and 5 in), whereas tablets exhibit a diagonal screen size upward of approximately 20.32 cm (8 in). In recent years, both classes of devices have exhibited similar components and features—making the primary distinguishing feature the greater screen real-estate of the tablet. That is, tablets are able to display more data points to the user simultaneously, without scarifying portability.

This is in contrast to traditional hardware such as desktops or laptops which are bound to a physical location—generally an office. In the case of the industry partner, had decision support information been made available on a personal computer, floor managers and operators would have needed to traverse between their on-floor location and an office to obtain the necessary information.

D.1.3 Scope exclusions

Factors that were not considered include:

- User authentication and authorisation (all connections were authorised).

Table D.1: Functional requirements of the mobile client.

Category	No.	Feature	Req. No.	Description
Data acquisition	1	Create a job.	FR_M01	Add new jobs as they arrive at the system.
	2	Create a task(s) associated to a job.	FR_M02	Add a single or set of tasks and associate them with a job.
	3	Associate a task with an RFID card.	FR_M03	Associate an RFID card with a task.
	4	Update job priority.	FR_M04	Update job priority on expedited orders.
Decision support	5	Display all jobs.	FR_M05	Fetch and display all jobs in the system (new, WIP, and completed).
	6	Display all tasks.	FR_M06	Fetch and display all tasks in the system, including their queue positions.
	7	Display all operations.	FR_M07	Fetch and display all operations and their queue contents.
	8	Display global dispatching rule.	FR_M08	Display the selection discipline recommended by the SOE.
	9	Display task rankings.	FR_M09	Display the relative rank of each task.
	10	Trigger SOE call.	FR_M10	Manually invoke the SOE to trigger a simulation optimisation.
	11	Display job expected completion times.	FR_M11	Display each job's expected completion time (simulation response).

- Session handling (no session key was maintained between the server and client).
- Caching (on-device caching was not implemented, see D.1.4)

For real-world deployment these exclusions and assumptions should be addressed.

D.1.4 Notes on caching

Client and server side caching was not considered in this thesis due to time constraints. This area should looked into for a real-world reactive scheduling system design as caching has the potential to reduce server load and the number of client-side requests. This is achieved by storing response data in a fast ephemeral store, or cache. If the state of the requested data has not changed, that data are served from the cache, preventing either a computational task or a database read. Similarly, client-side caching may eliminate the need for the client to perform certain Web service calls altogether, improving battery life (power-hungry cellular or WiFi radios do not need to be powered up) and minimising bandwidth usage.

If implemented incorrectly, caching may introduce inconsistencies. For example the client may request a set of data from the server but is instead returned a cached version of the data that is out-of-date. Given the nature of the necessity for up-to-date information on the factory floor, caching would need to be implemented carefully.

D.2 Conceptual diagrams

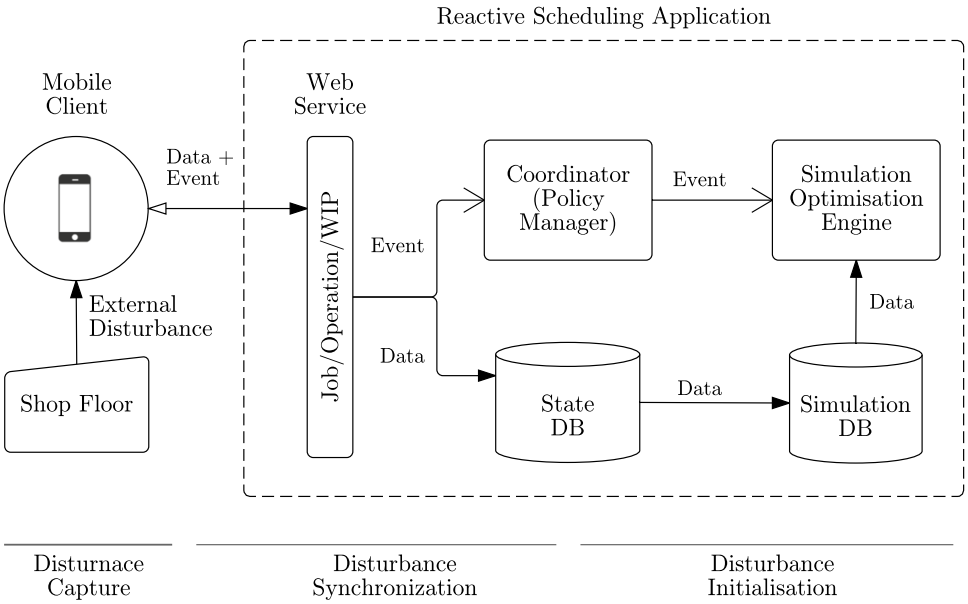


Figure D.1: Conceptual architecture — Data acquisition.

D.3 UML modelling diagrams

Table D.2: Use case scenario: Adding a job to the system.

Use case name:	Add a job
Unique Id:	MC_001
Category:	Data capture
Actor(s):	Floor manager
Level:	Blue
Description:	Allow floor manager to add a new job to to the system.
Triggering event:	A new job arrives at the system boundary.
Trigger type:	External, Temporal

Steps Performed (Main Path)	Information Required
1. User navigates to the 'add job' screen.	N/A
2. User taps 'Add'.	N/A
3. User capture job details.	Gauge code, quantity, priority, release date, due date
4. User adds one or more tasks to the job.	Number of tasks, Batch Size
5. User associates a task with an RFID card.	RFID card unique identifier

Preconditions:	The app is in the 'home' state. All 'add job' dialogs have been closed.
Postconditions:	Floor manager has successfully added a job to the system.
Assumptions:	Floor manager has access to all required information to capture the job.
Success guarantee:	Job has been added to the system.
Minimum guarantee:	Participant was able to logon.
Requirements met:	Floor manager was able to add a job to the system.
Outstanding issues:	How should rework be handled?
Priority:	High
Risk:	Medium

Table D.3: Use case scenario: View job estimated completion times.

Use case name:	Job estimated completion times
Unique Id:	MC_002
Category:	Decision support
Actor(s):	Floor manager
Level:	Blue
Description:	Floor manager views the expected completion time of a job.
Triggering event:	Floor manager view the list of jobs in the system.
Trigger type:	External, Temporal

Steps Performed (Main Path)	Information Required
1. User navigates to the 'jobs' screen.	N/A
2. User taps on the job.	Job Id

Preconditions:	All user dialogs boxes are closed. At least one job has been added to the system.
Postconditions:	Floor manager has successfully viewed to the expected completion time of jobs in the system.
Assumptions:	At least one job is present in the system. The SOE has executed at least once.
Success guarantee:	Job has been added to the system.
Minimum guarantee:	Participant was able to logon.
Requirements met:	Floor manager is able ot view the job's estimated completion time.
Outstanding issues:	How should rework be handled?
Priority:	High
Risk:	Medium

Table D.4: Use case scenario: Trigger SOE.

Use case name:	Manually Trigger SOE
Unique Id:	MC_003
Category:	Decision support
Actor(s):	Floor manager
Level:	Blue
Description:	Floor manager manually overrides the SOE. The simulation experiment is without a rescheduling point having been reached.
Triggering event:	Manager decides that the data must be refreshed.
Trigger type:	External

Steps Performed (Main Path)	Information Required
1. User navigates to the settings screen.	N/A
2. User taps 'Trigger SOE override'.	N/A
3. User confirms action on UIAlert prompt.	N/A

Preconditions:	The app is in the 'home' state. All 'add job' dialogs have been closed.
Postconditions:	Floor manager has successfully triggered the SOE and a simulation experiment has run.
Assumptions:	At least one job is in the system. The simulation model is able to initialise off of the snapshot.
Success guarantee:	SOE has returned the new dispatching rule.
Minimum guarantee:	Web service registered SOE event.
Requirements met:	Floor manager was able to manually execute and optimisation call.
Outstanding issues:	N/A
Priority:	High
Risk:	High

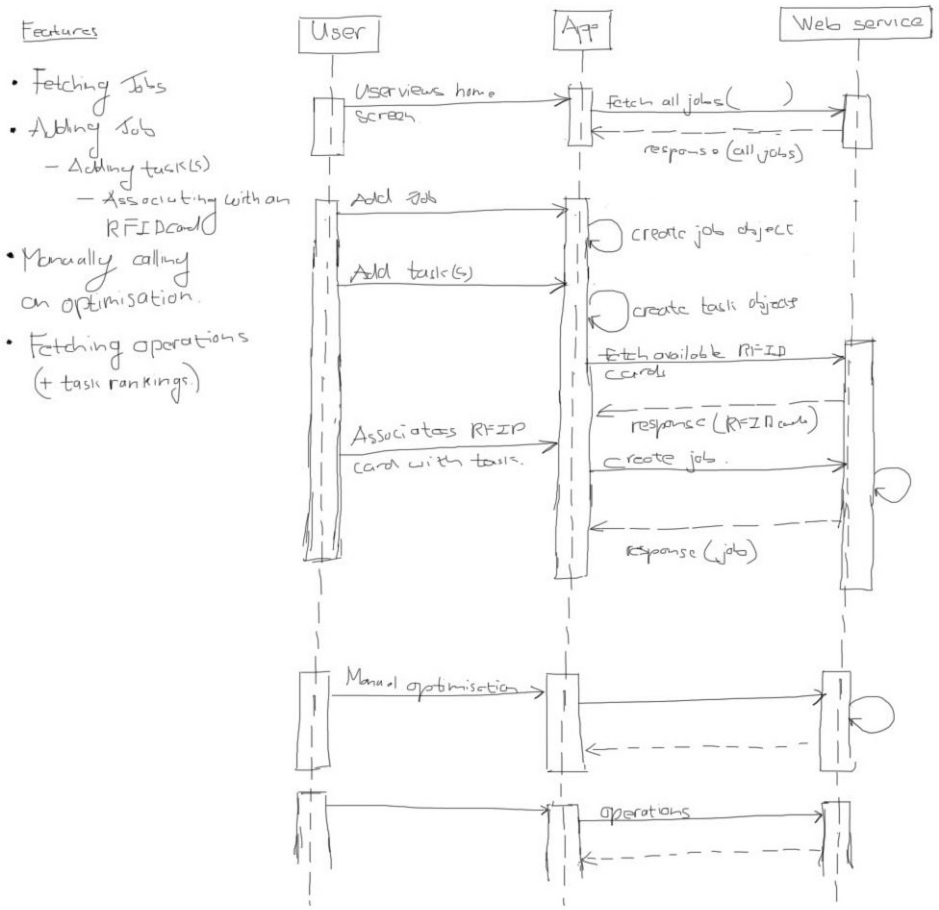


Figure D.2: Sequence diagram: Mobile client first iteration.

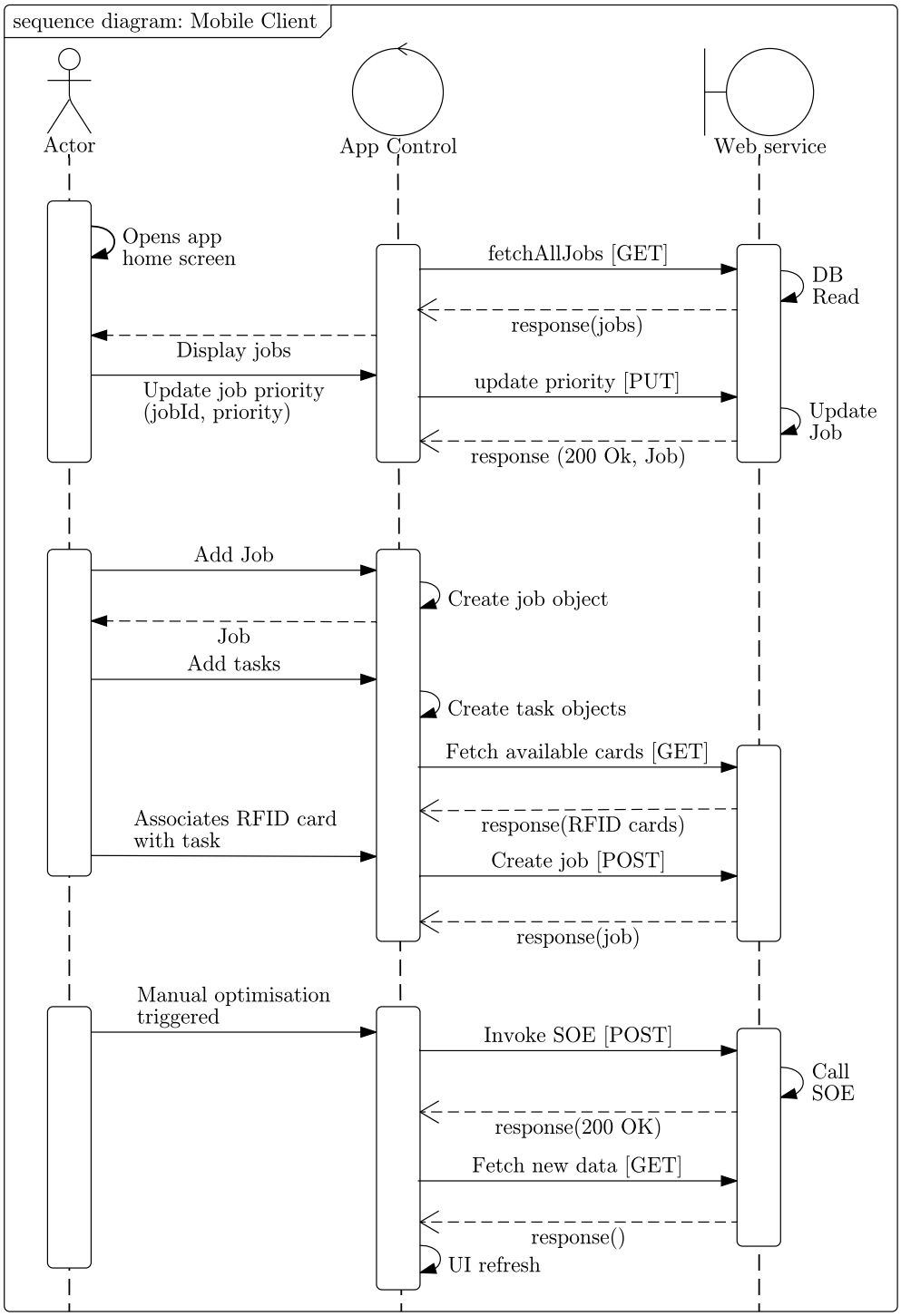


Figure D.3: Sequence diagram: Mobile client final iteration.

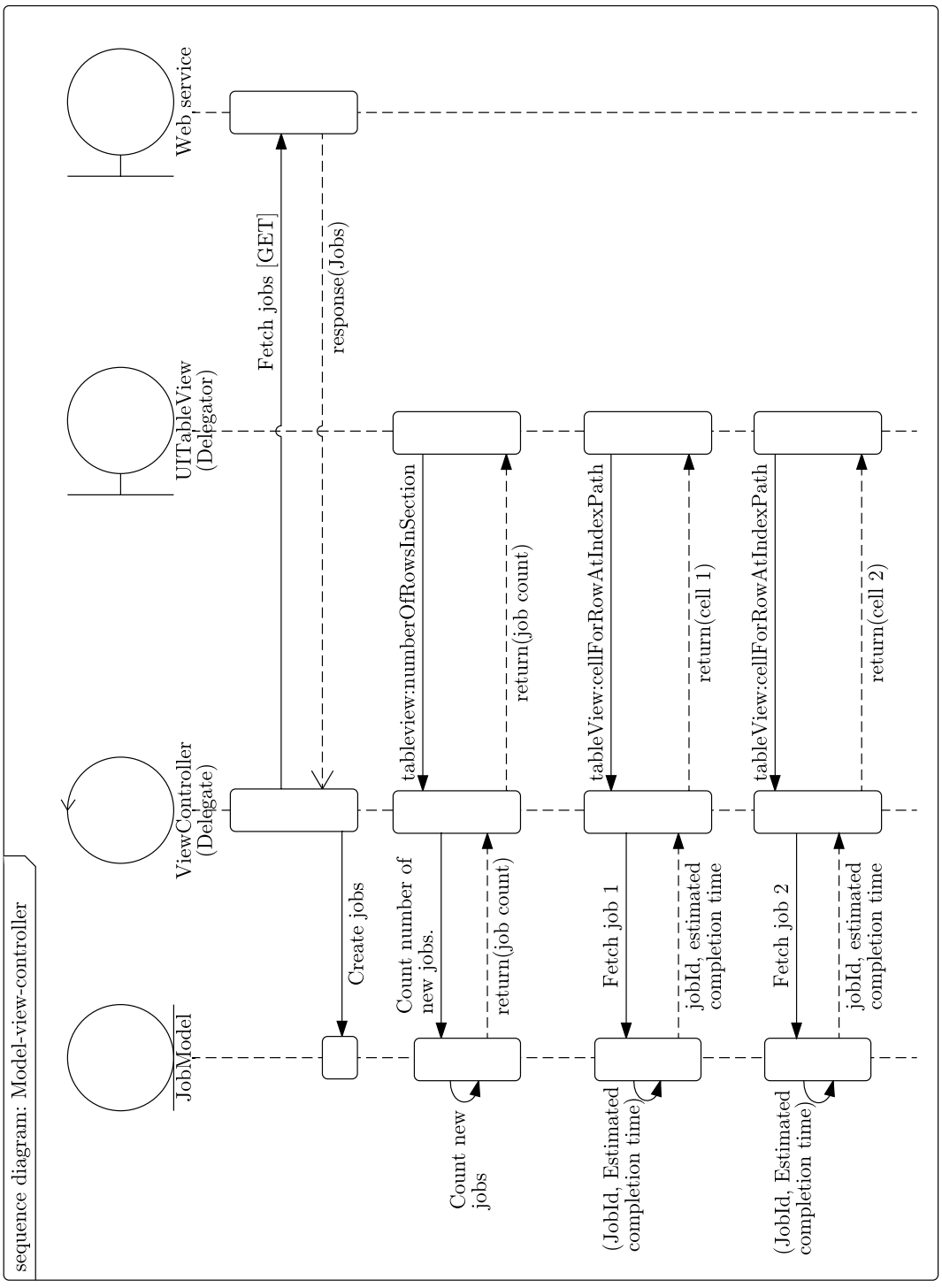


Figure D.4: Sequence diagram demonstrating delegation and the MVC design pattern.

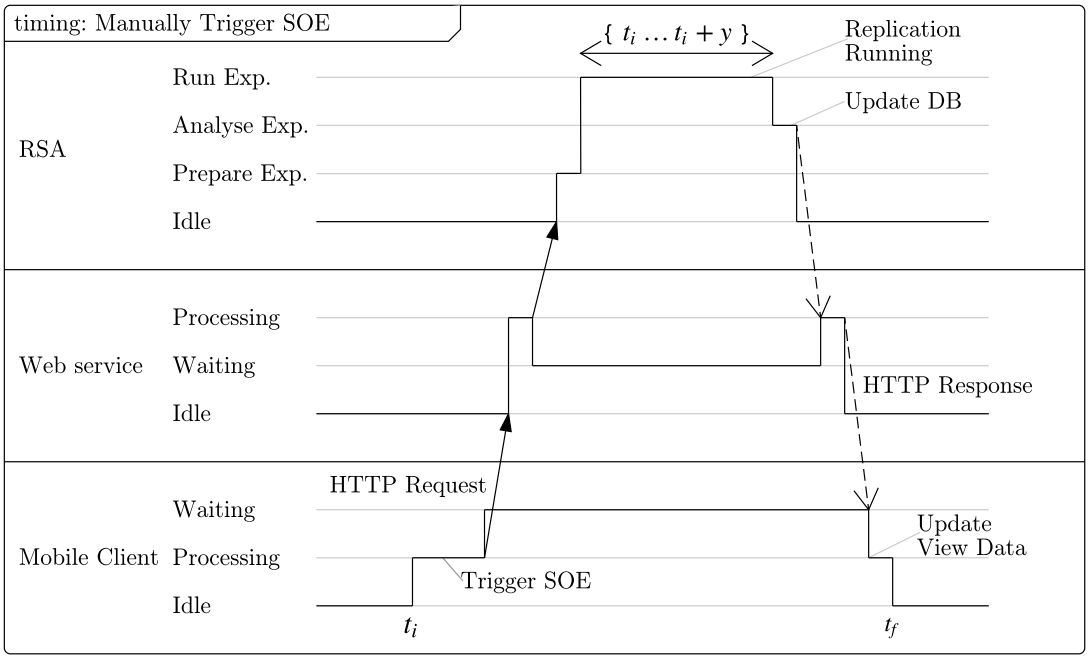


Figure D.5: Timing diagram: Manually trigger SOE.

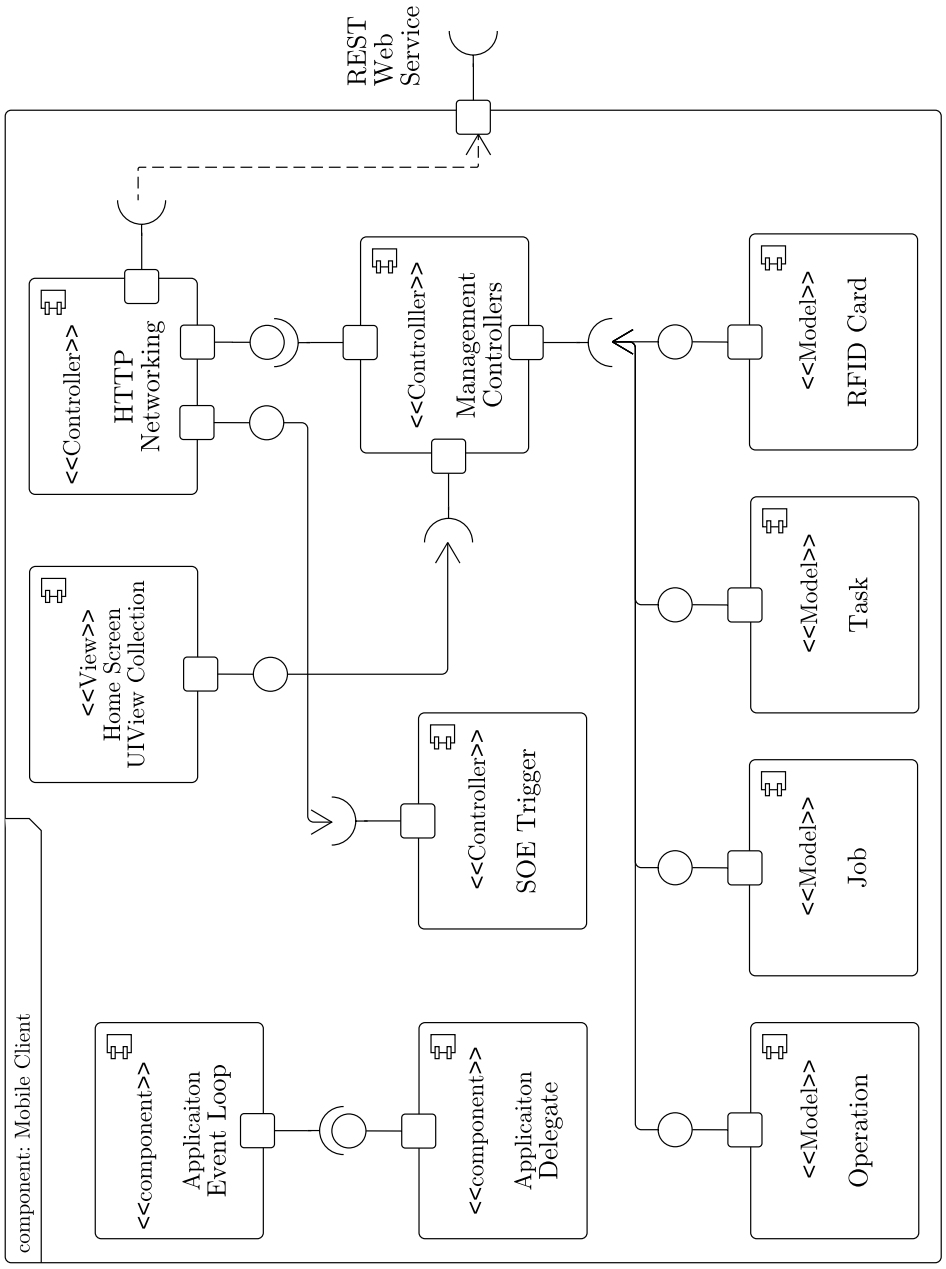


Figure D.6: Component diagram: Mobile client.

D.4 User interface mock-ups

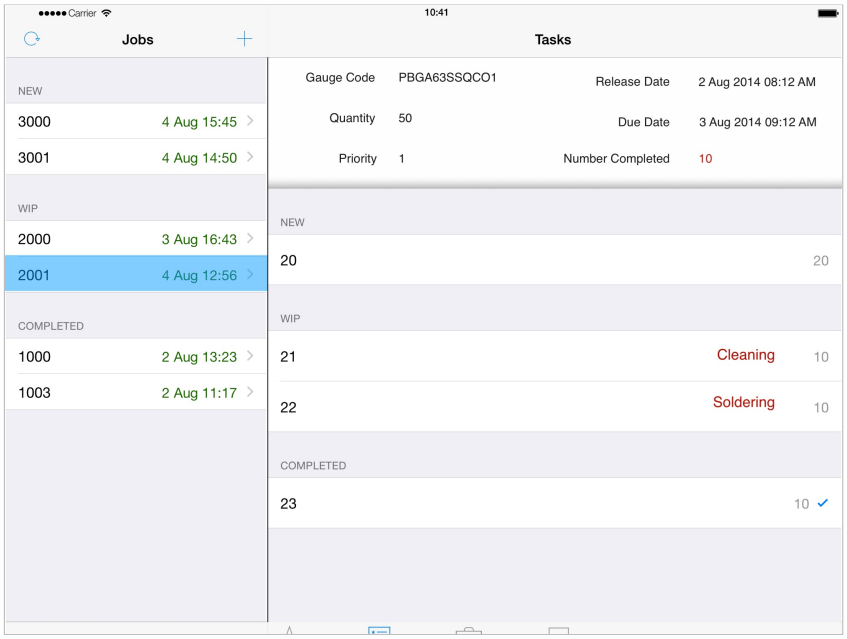


Figure D.7: Home screen mock-up.

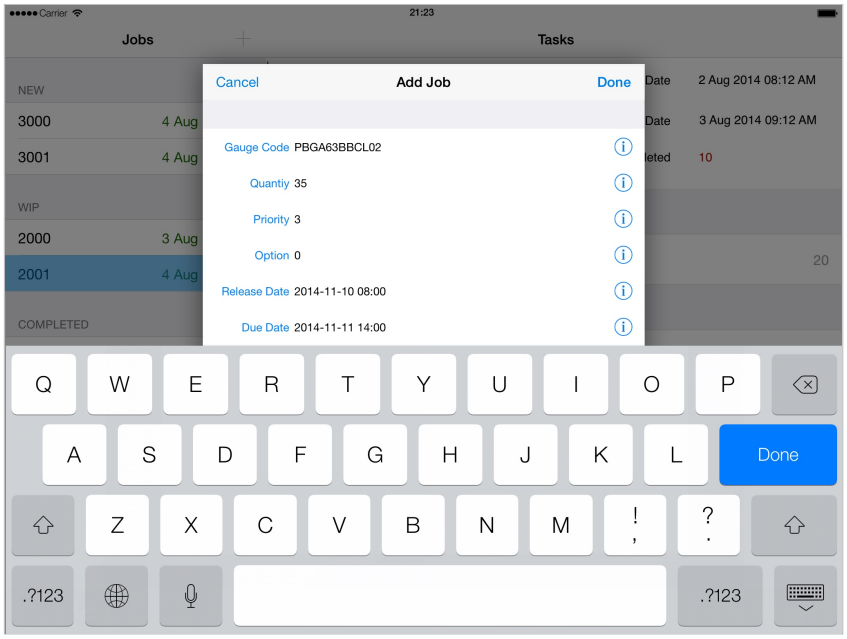


Figure D.8: Add a job screen mock-up

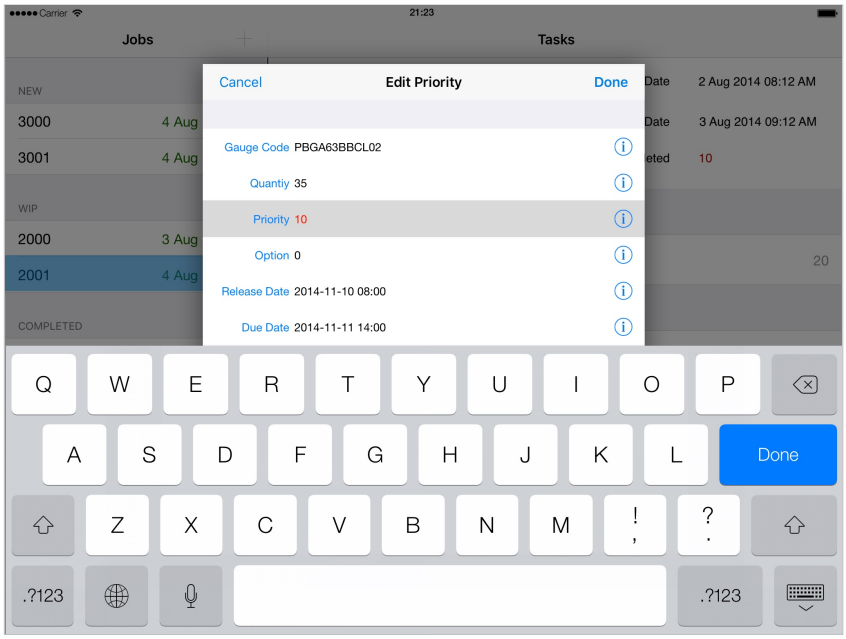


Figure D.9: Adjust job priority screen mock-up.

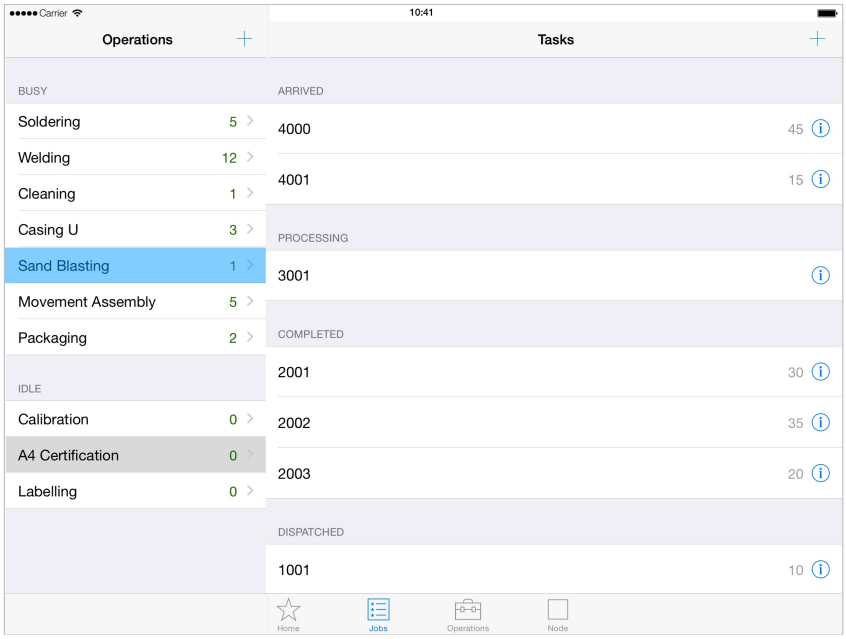


Figure D.10: View workstation load screen mock-up.

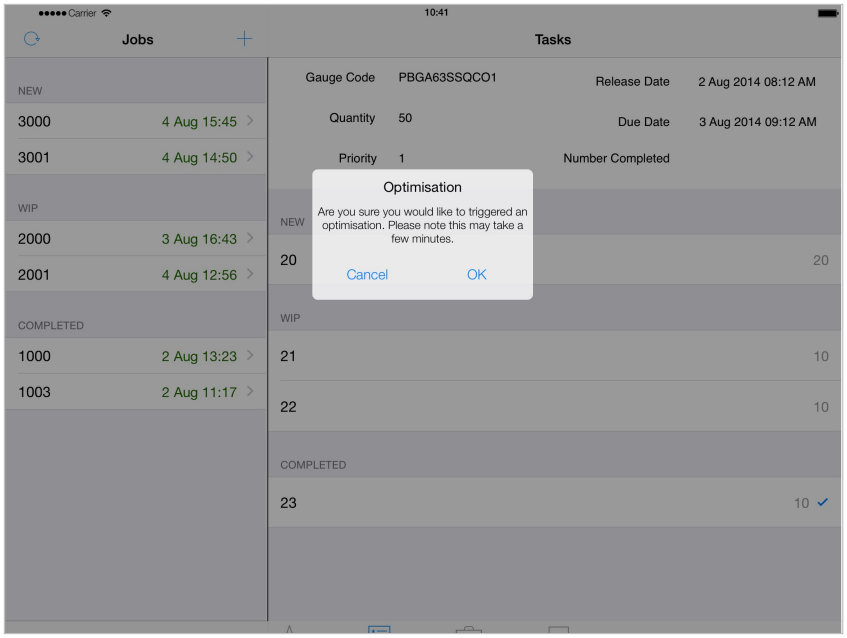


Figure D.11: Mockup screen: Manual invocation of the SOE.

Appendix E

Reactive scheduling system

E.1 Supplementary material

Supplementary material on the development of the reactive scheduling application.

E.1.1 Design patterns used in design and development the RSS

Arriving at good code meant following best practices. The following design patterns were employed in the construction of all software in this thesis.

E.1.1.1 Object orientated programming

The four fundamentals of object orientated programming (OOP) were adhered to through the design and construction phases:

- Encapsulation: Public and private implementation details were separated, e.g the simulation model only exposed those methods necessary for its control by the SOE.
- Inheritance: Class inheritance improves code readability and enables the reuse of functionality.
- Abstraction: Modelling the essence of an object without adding unnecessary detail.
- Polymorphism: One name, many forms. methods with the same name but different underlying functionality.

These OOP fundamentals helped manage the software complexity by applying key concepts:

- Ideas were abstracted into software objects making them easier to understand and maintain.
- Applications were organised in natural components.
- Functionality, e.g. SOE was encapsulate to allow understandable related data and logic to be kept together.

These concepts were critical to making sense of the interaction between the simulation model and the reactive scheduling application. Furthermore, the application of the principles that underlie object-orientated analysis and design tend to result in architectures that poses the desired properties of organised complexity (Booch, 2007).

E.1.1.2 Dry

The principle of DRY, or "don't repeat yourself", was used throughout the software development stage. This simple principle affords a number of advantages:

1. Reduction in repetition, especially for layered systems such as a reactive scheduling system.
2. Simplification of maintenance and debugging as functionality is guaranteed to be specified in a single location.

The principle of dry aided in constraining functionality to a single location.

E.1.1.3 Top down

The unified process does not specifically call for the top down approach, however the principles are implicit in its approach. Principles include (Kendall & Kendall, 2011):

1. Avoiding the chaos of attempting to design a solution all at once.
2. Allows work to be separated and placed into parallel design and development streams. A critisims of bottom up approaches is that work units become blurred and become incorrectly categorised.

These principles assisted creating an architecture the met its design properties of: scalability, maintainability, modularity.

E.1.2 Advantages of using a Web service

Some typical advantages of implementing a Web service are listed below:

1. Security: User permissions management is well developed in the service layer. Harder to control and not well supported in the database layer.
2. Scalability: Server side scaling is well supported. Distributing request across parallel databases is easily handled in the service layer.
3. Encapsulation: Modification of the underlying database may occur without necessarily affecting Web service endpoints, and therefore no modification of the client application is required.
4. Client independence: Support of the standard HTTP 1.1 protocol is all that is required for a client device to interact with a Web service. The manufacturer, device type, software platform, etc. are largely irrelevant during service.
5. Versioning: Multiple versions of a service may be supported simultaneously. Thus, support for clients that interact with prior versions of the service can be maintained and without code modification.
6. Centralisation: Application logic may be changed on the server once, as opposed to updating every client application.

E.1.3 Peer-to-peer versus client-server

It is conceivable that an RSS could be entirely orchestrated by intelligent front-end devices (such as tablets or smartphones) in a peer-to-peer topology—negating the need for a central server. This solution, though removing the complexity of the back-end, would significantly escalate the complexity of designing the front end. Without a centralised coordinating server, a number of complications would arise, including:

- The computational requirement, particularly when considering the demands of running a simulation experiment, may exceed the capability of front-end devices.
- Web services, exposed off application servers, are typically device and platform agnostic. A peer-to-peer arrangement would force clients to interact using a device or platform specific protocol.
- The increased processing and communication demands negatively impact on the devices battery life—a non-trivial factor when designing for mobile.

- Data are difficult to backup as there is no single central store.
- Operating on data server-side is generally done through industry standard languages such as SQL. The landscape of front-end device persistent stores is still varied.

Client-server architectures are superior to pure peer-to-peer architectures in all but a few specific cases. This has led to client-server being the dominant architecture employed by most enterprises.

E.1.4 Revised cloud architecture

The network architecture diagram of Figure E.15 shows a component-based 'monolithic' architecture, i.e. single central server. The principles of SOA advocate that the capabilities of large applications should, over time, be separated out to form a distributed service-orientated architecture. In Section E.1.4 a distributed architecture is explored which could potentially be used for a future reactive scheduling system.

Figure E.1 shows an alternative service-based architecture to the monolithic architecture of Figure E.15. The main distinguishing factor of the proposed architecture is that its components are distributed, i.e. servers are separated by a network and communicate by exchanging messages. As shown in Figure E.1, the RSA was partitioned into three services. The Web service was extracted to a load balanced presentation layer, and replication running to a load balanced worker role. This arrangement has the potential to improve certain system properties, such as scalability for instance. By running replications on a dedicated replication server, new instances of the replication server could be spun up prior to experiment execution, and spun down upon completion (horizontal scaling). This is in contrast to a monolithic deployment where the server is forced to remain on-line as it supports many other functions in addition to replication running. Furthermore, since the computational requirements of executing a simulation model are typically higher than the requirements of the RSA (event management, updating queue positions, etc.), replication servers could be highly optimised for executing a simulation model, leading to a better experiment cycle time.

Apart from scalability, distributed architectures also allow for parts fo the system to be upgraded without the need for recompiling and redeploying the entire solution (as was the case with the application server developed in this thesis). The advantages of a distributed architecture come with some drawbacks however, predominantly related to solution complexity. For example, the definition of interfaces between networked components (such as the REST interface described in Section 7.3.2) is generally more

involved[1] than specifying inter-component communication with a single application server. Nevertheless, such architectures hold potential and thus have become a popular research topic and the subject of numerous articles.

It is worth noting that the design of a reactive scheduling system backend is possible using the techniques of SOA alone. The described components, e.g. data manager, could be extracted from the monolith and placed in an environment of their own. This transition would require each component to transform into a service through the addition of a service layer—allowing it to communicate in accordance with a service contract over a network. The advantages of this architecture include isolated deployments and efficient scalability for each service. Despite these advantages, the networking framework needed to arrive at this architecture were deemed too time consuming for this thesis and is left for future research.

E.1.5 Debugging

Debugging, as with all projects, was a continuous process. A technique that proved useful in this project was minimum viable example (MVE). In this technique, bugs that cannot be solved through compiler warnings or simple run-time tests are exposed by either:

1. incrementally reducing the functionality of the system until only the core code necessary to produce the bug remains, or

2. starting from scratch and adding code until the bug manifests.

MVE was found to be suitable for isolating issues when the code base was small, as with the sensor network. With larger code bases, the option 1 was still useful, but option 2 was found to be impracticable.

E.1.6 Security

Security, although not part of this research, is worth mentioning since it is an important factor for the real-world deployment of an RSS. Enterprises may wish to put systems in place to keep the data coming off the factory floor private. A hacker with access to the server or to the data stream as it moves over the network may, over time, be able to build up a detailed picture of the factory's operations and performance metrics. This

[1] Services must communicate in a stateless manner, a property which add complexity when compared to inter-object communication.

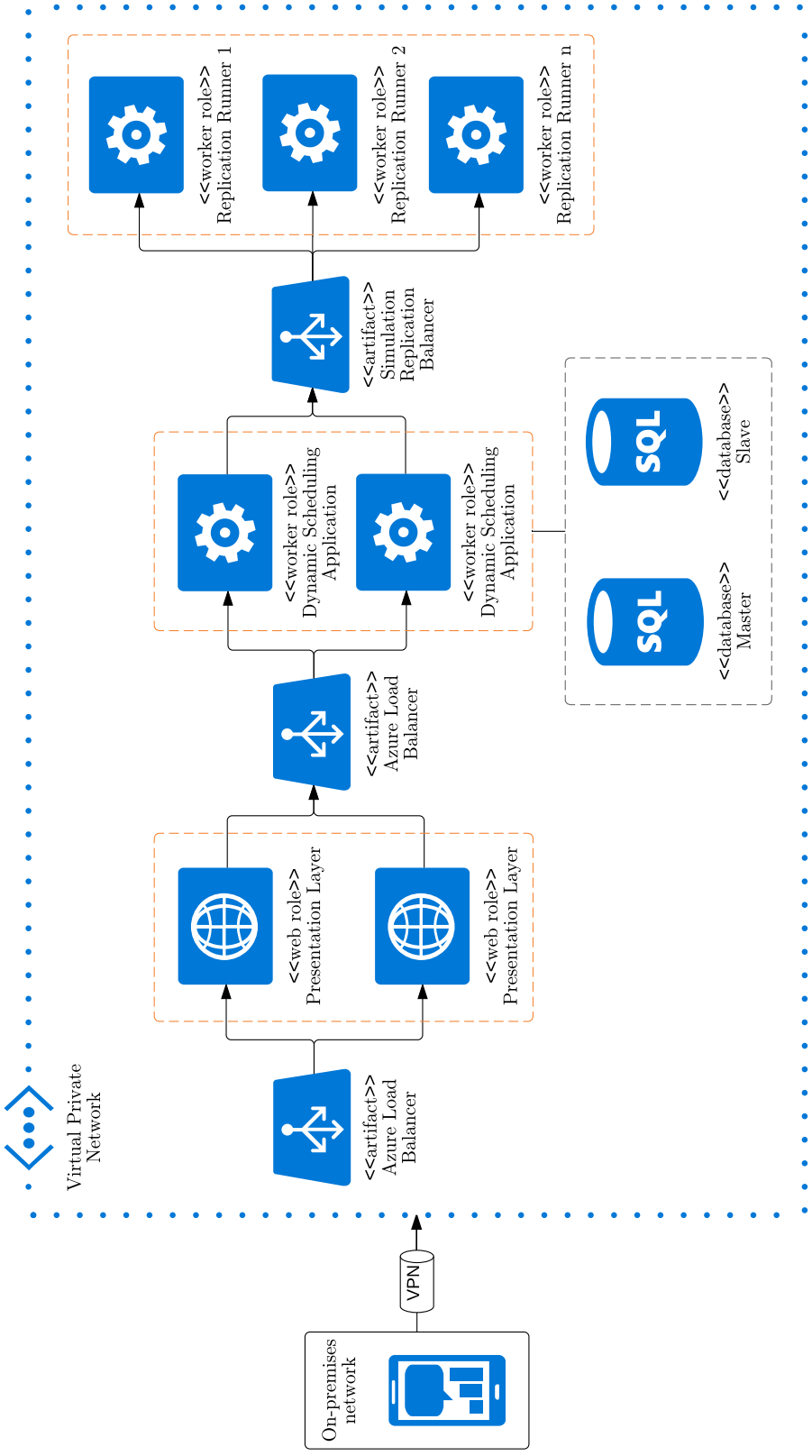


Figure E.1: Potential cloud architecture for the RSA.

data may then be sold to other competing enterprises—a tactic common on the internet today.

To prevent the interception of data, an end-to-end encryption solution, e.g. SLL, would need to be put in place. Data should be encrypted by the originating party (e.g. node) and then decrypted by the intended recipient (e.g. server)—thus preventing a man-in-the-middle attack. Server side security to prevent unauthorised database access is well documented and the reader is directed to the numerous books on the subject.

E.2 UML modeling diagrams

A package view of the this thesis showing the organisation of the software system.

Figure E.2 shows the top level packages that were to be developed.

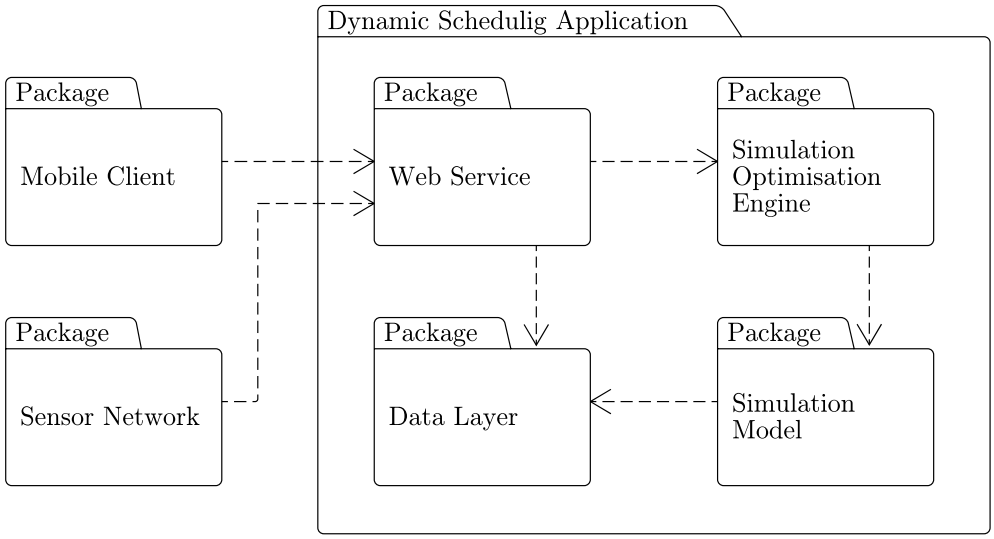


Figure E.2: Package diagram of the reactive scheduling system.

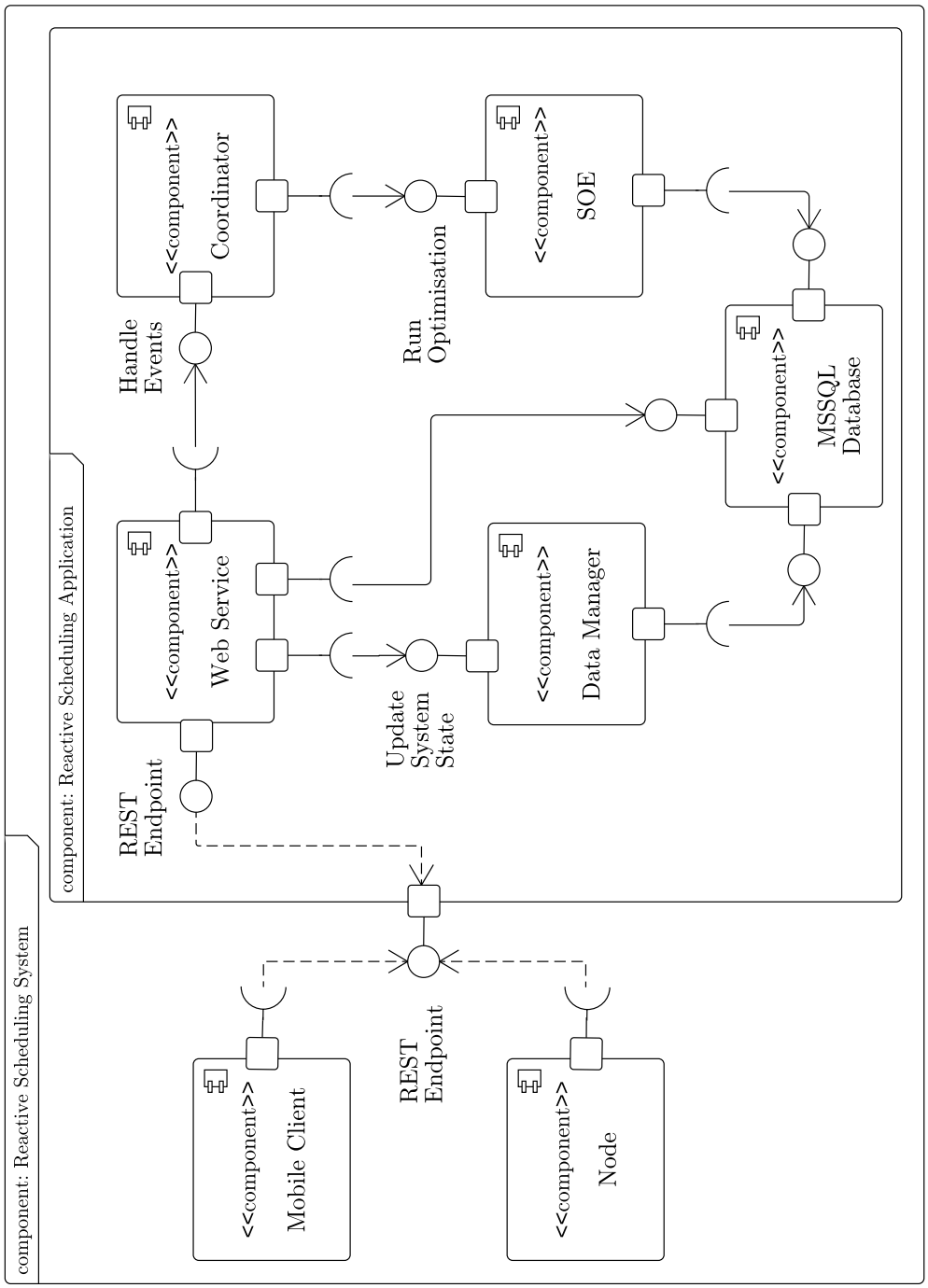


Figure E.3: Component diagram: Reactive scheduling system.

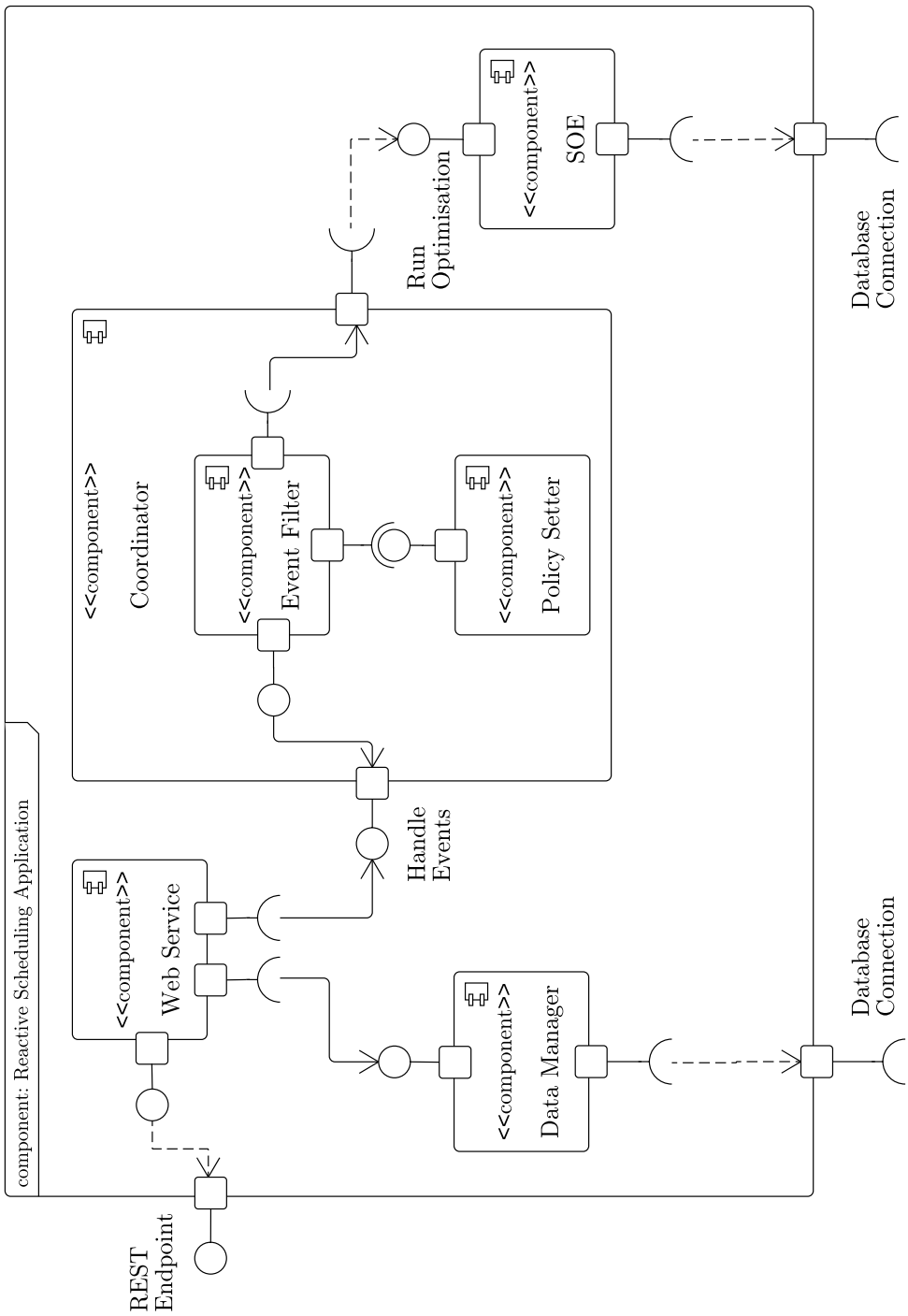


Figure E.4: Component diagram: Reactive scheduling application.

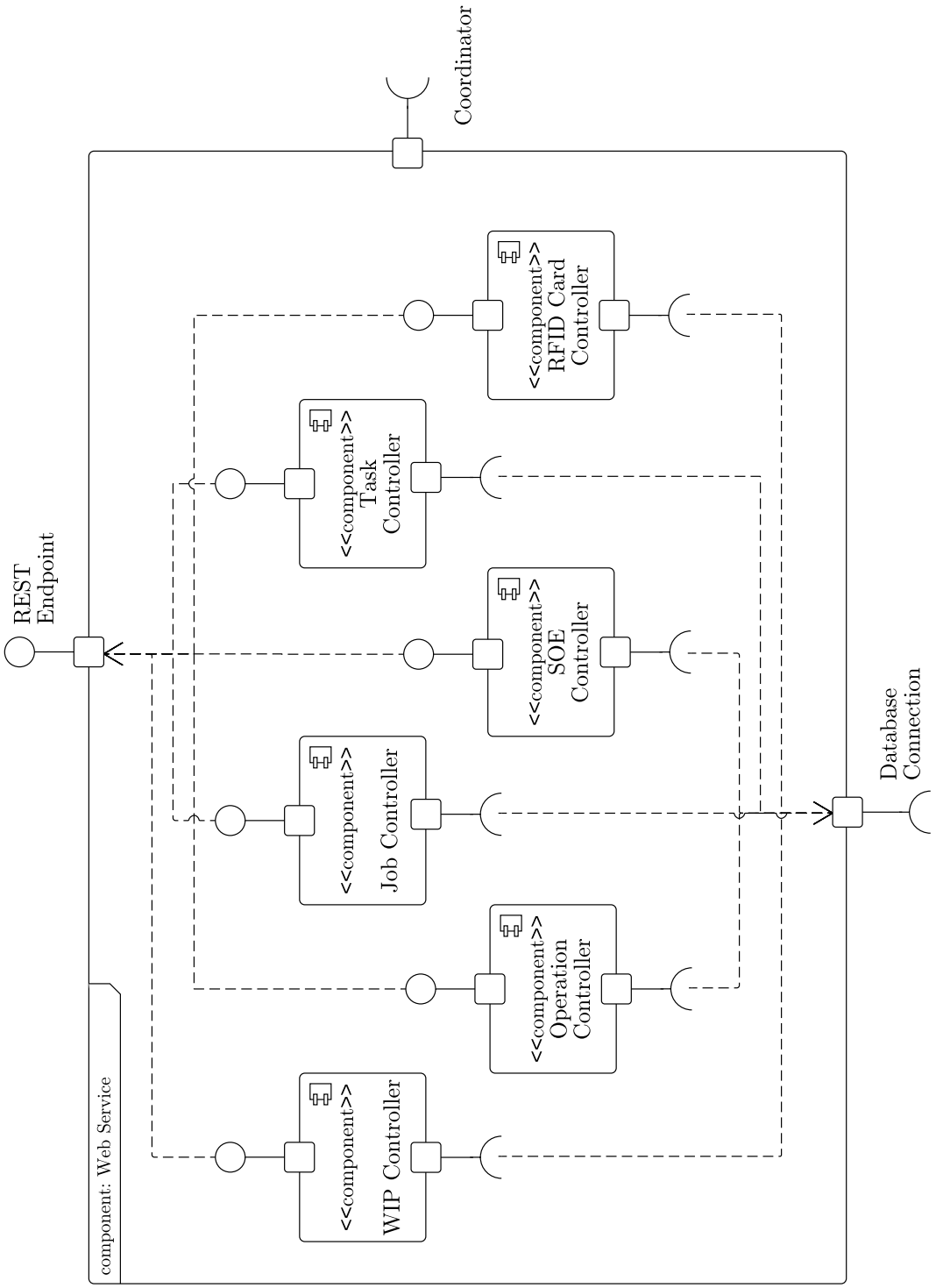


Figure E.5: Component diagram: Web service.

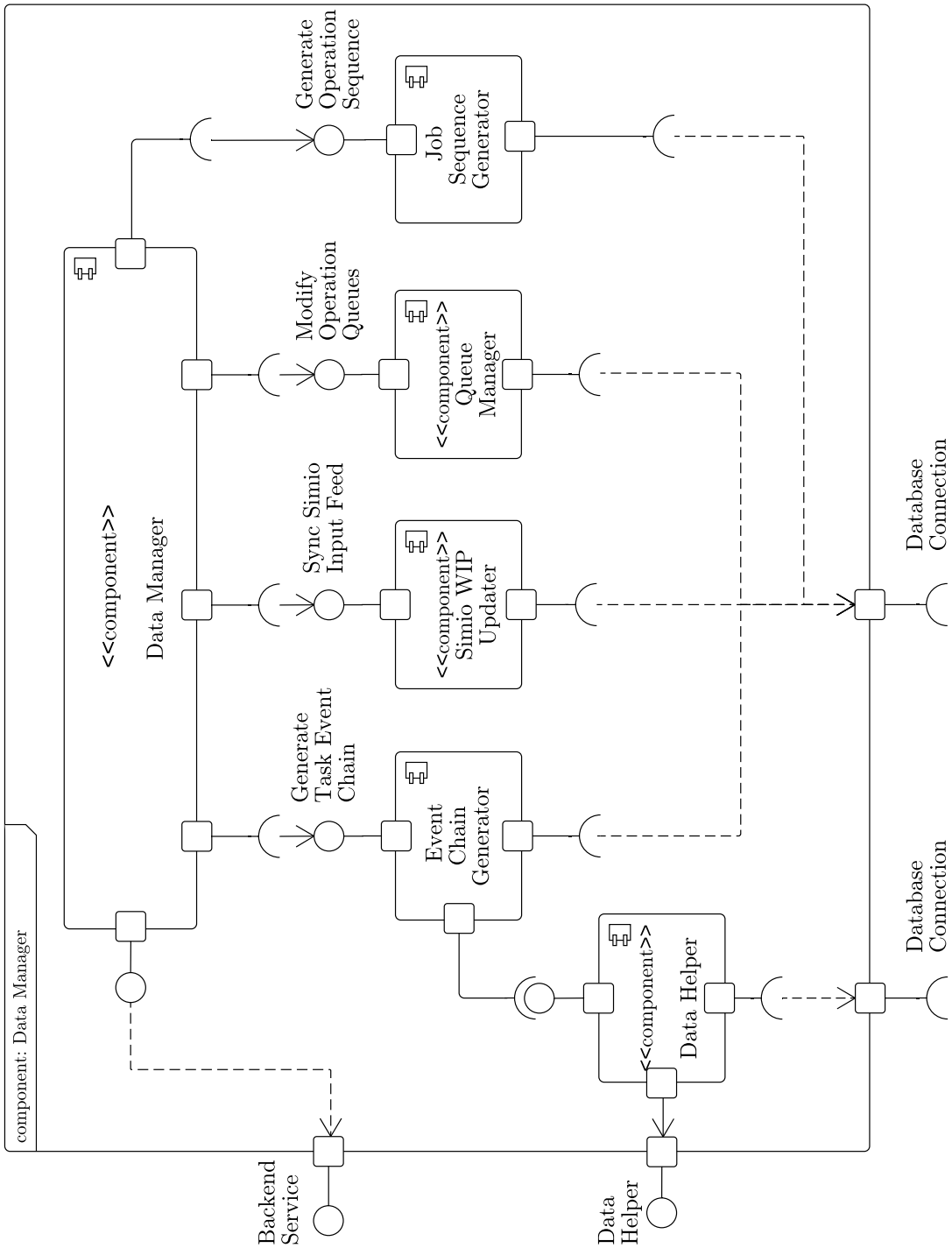


Figure E.6: Component diagram: Data manager.

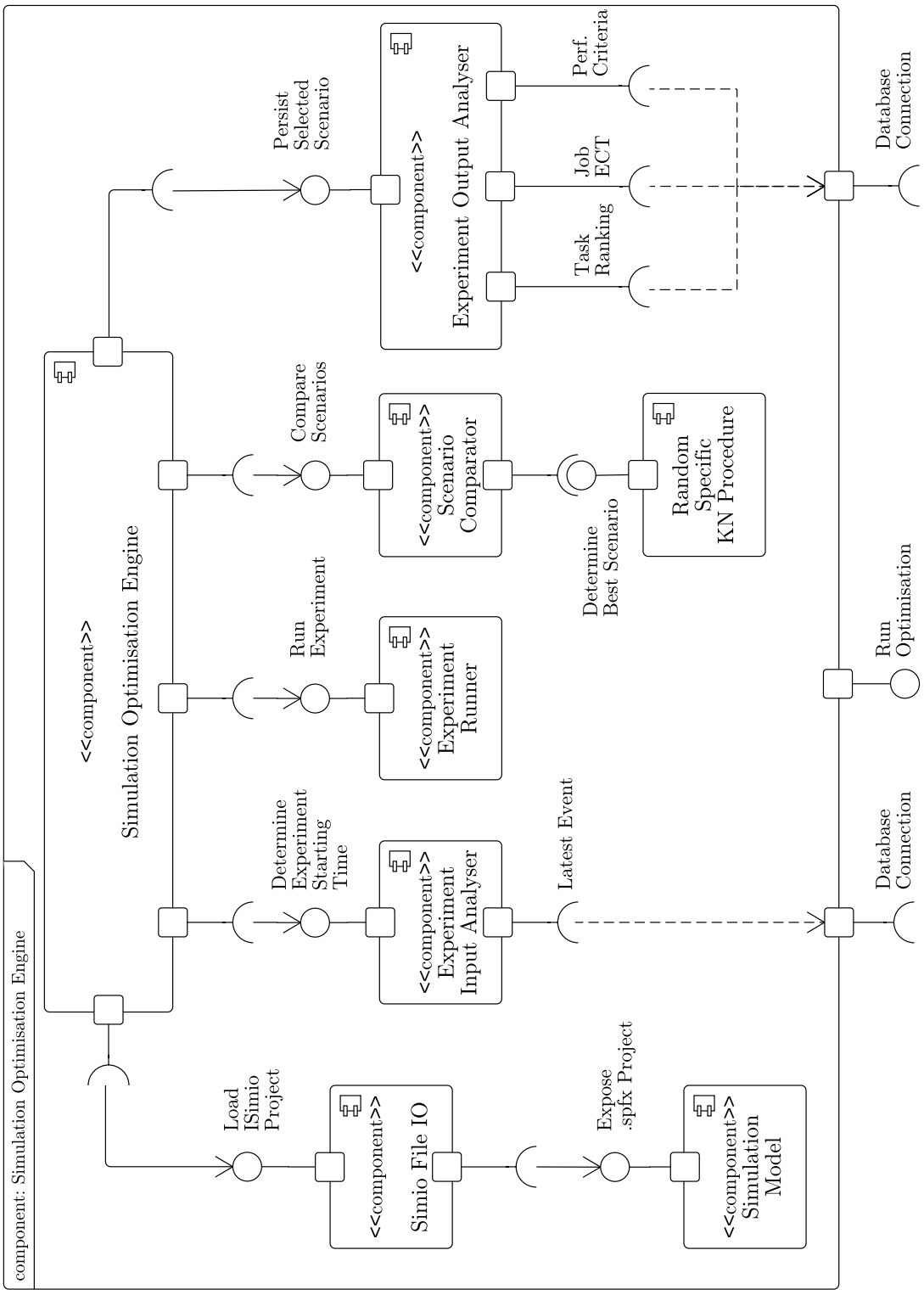


Figure E.7: Component diagram: Simulation Optimisation Engine.

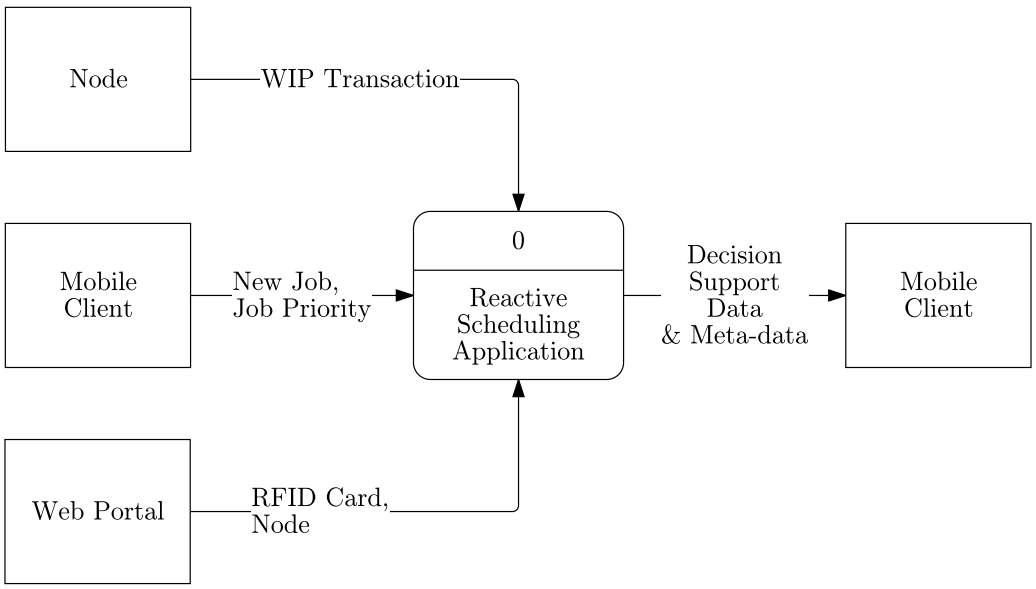


Figure E.8: Data flow diagram: Context — Reactive scheduling application.

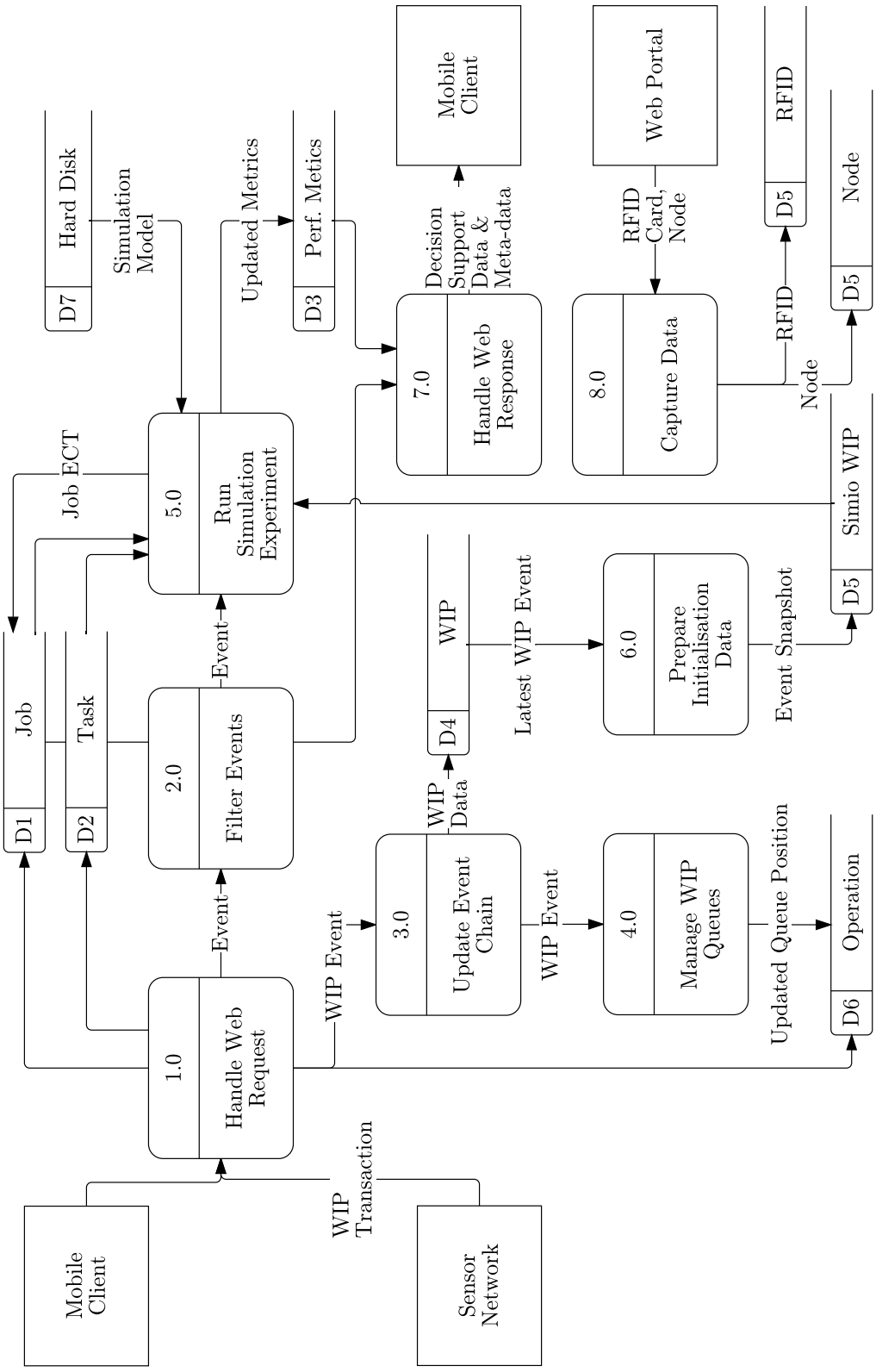


Figure E.9: Data flow diagram: Level zero — Reactive scheduling application.

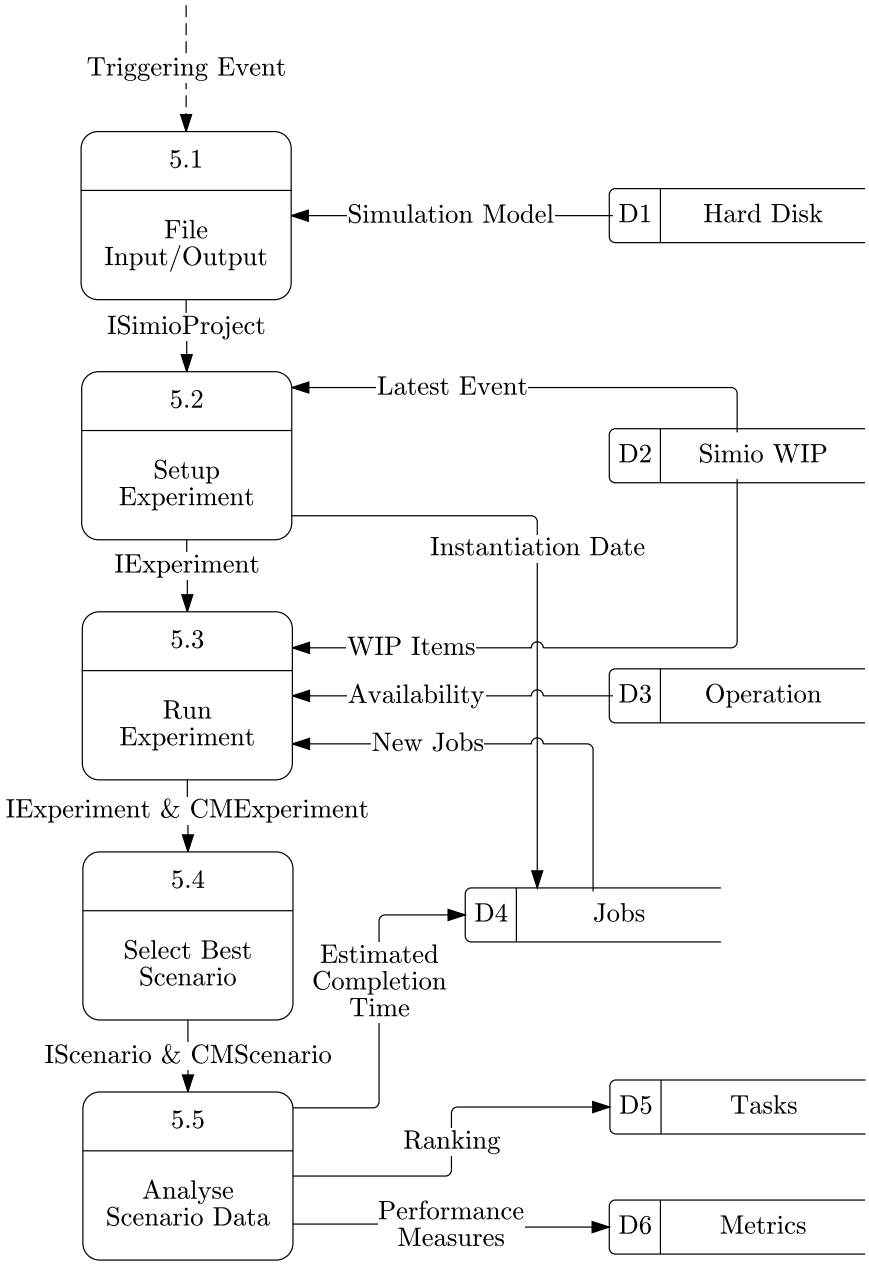


Figure E.10: Data flow diagram: Level one — SOE.

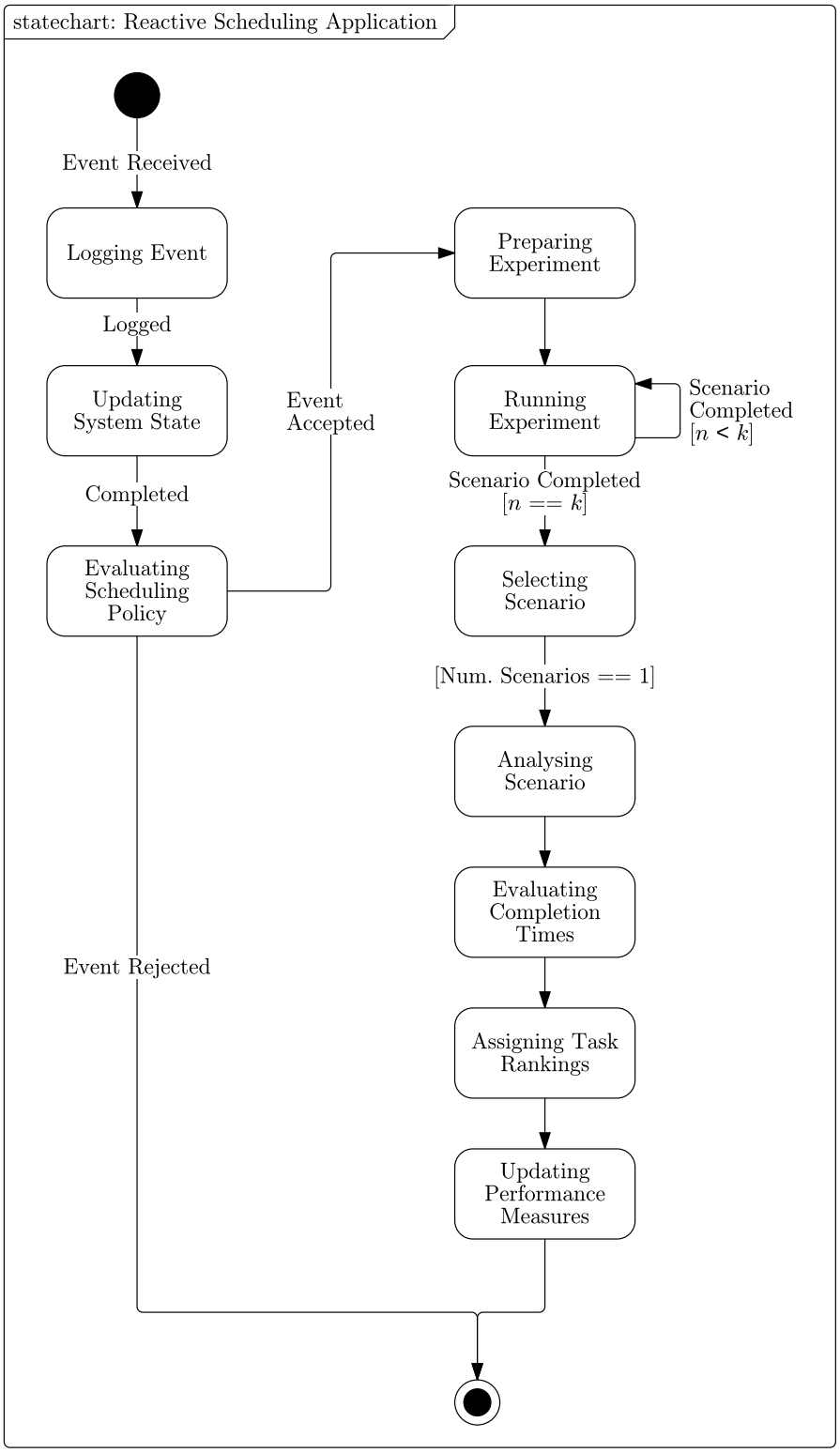


Figure E.11: Statechart diagram: Reactive scheduling application.

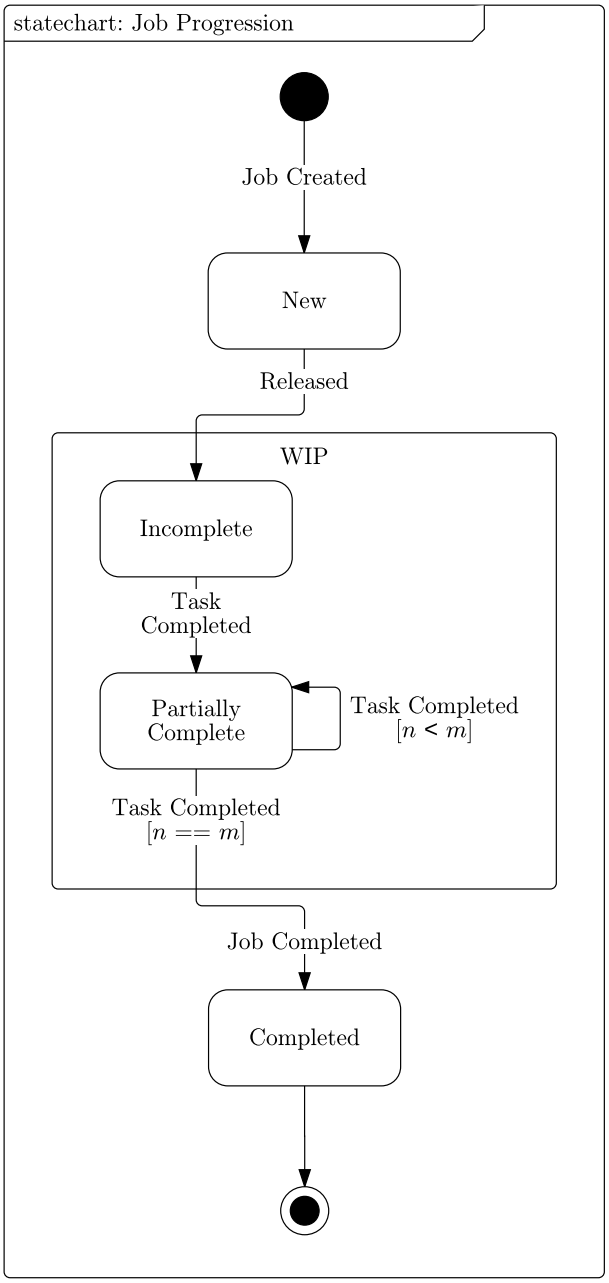


Figure E.12: Statechart diagram: Job.

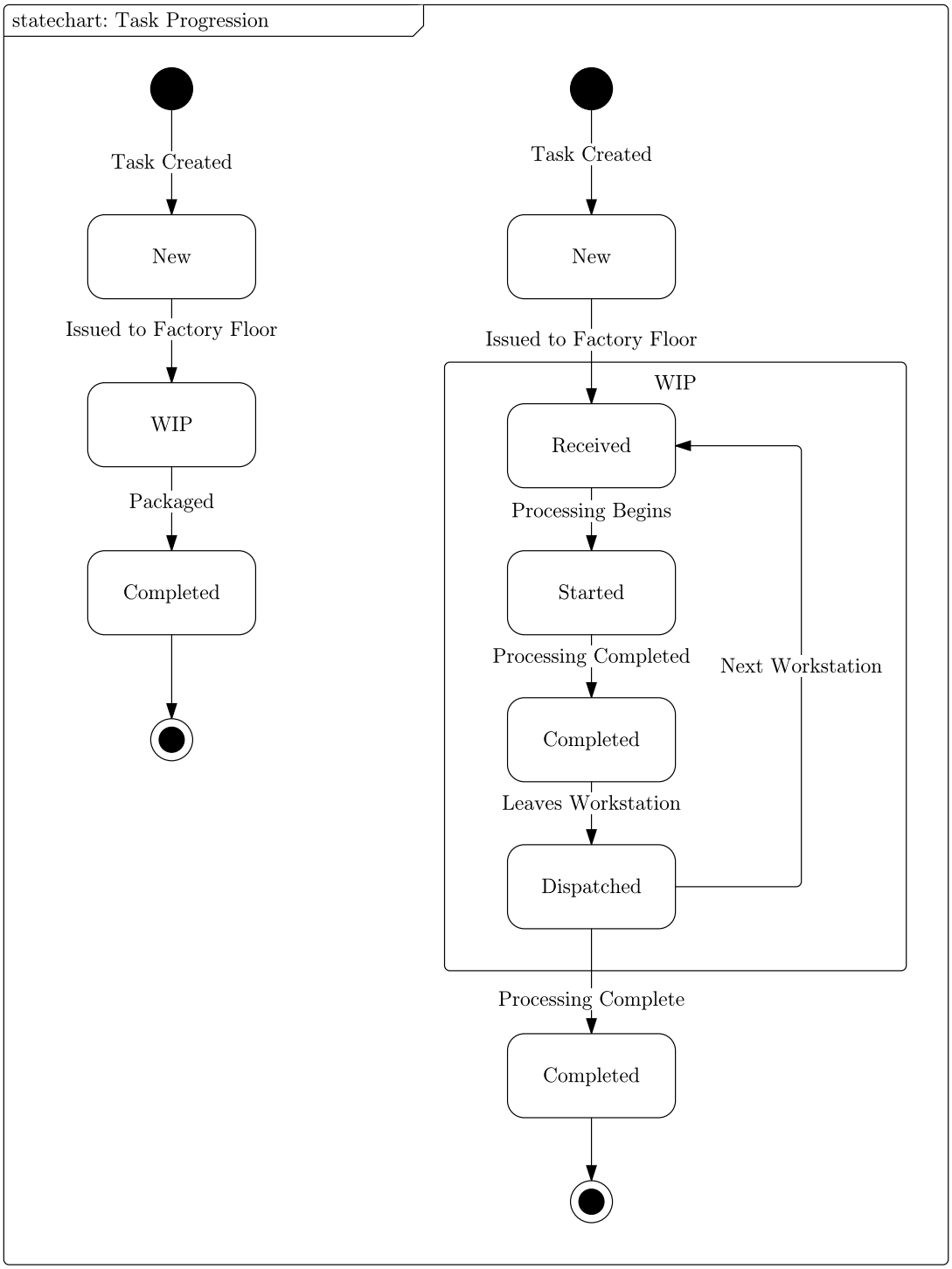


Figure E.13: Statechart diagram: Task.

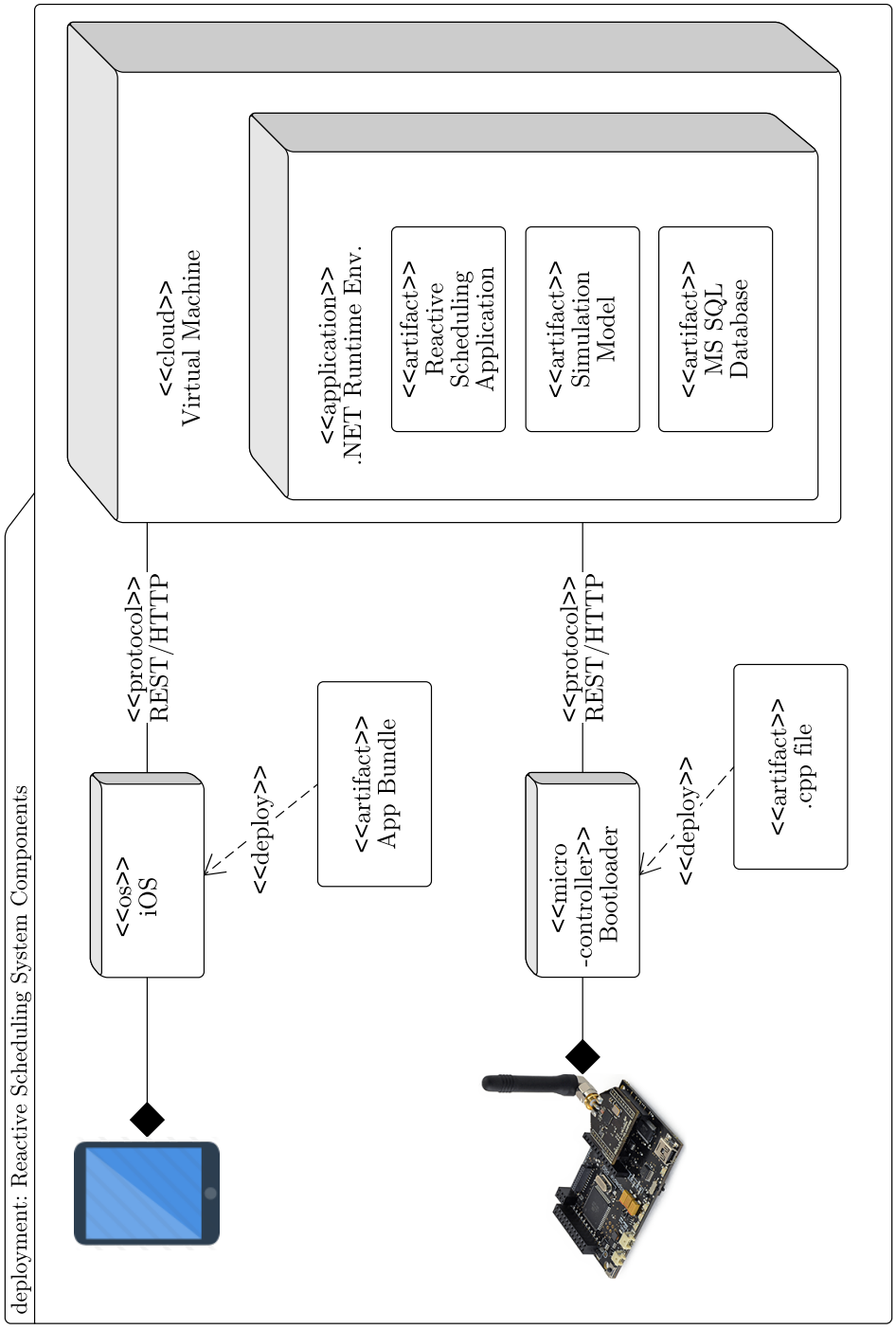


Figure E.14: Deployment diagram: Reactive scheduling system.

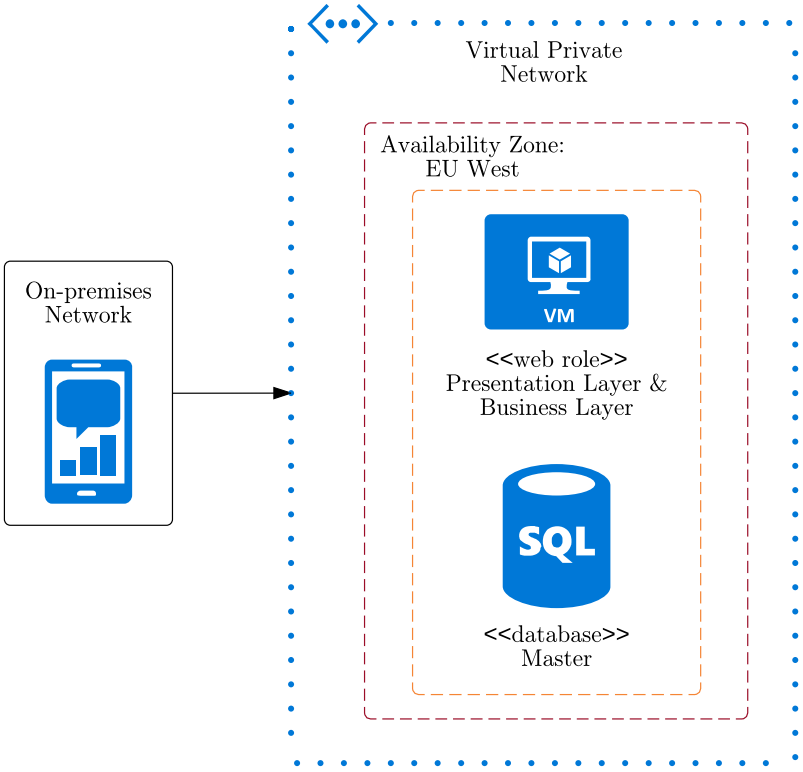


Figure E.15: Network architecture diagram of the reactive scheduling system (Microsoft® Azure cloud architecture notation).

E.3 Physical architecture

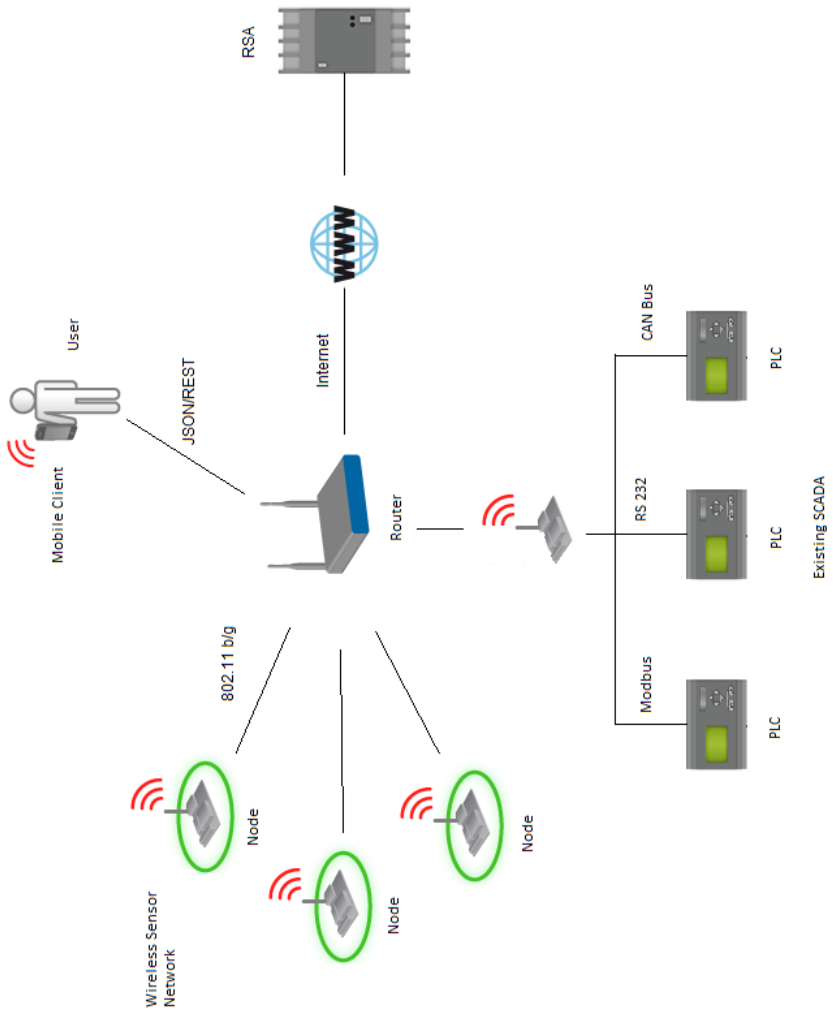


Figure E.16: Hardware and communication protocol of the RSS. A potential industrial backbone integration is also shown.

E.4 Miscellaneous

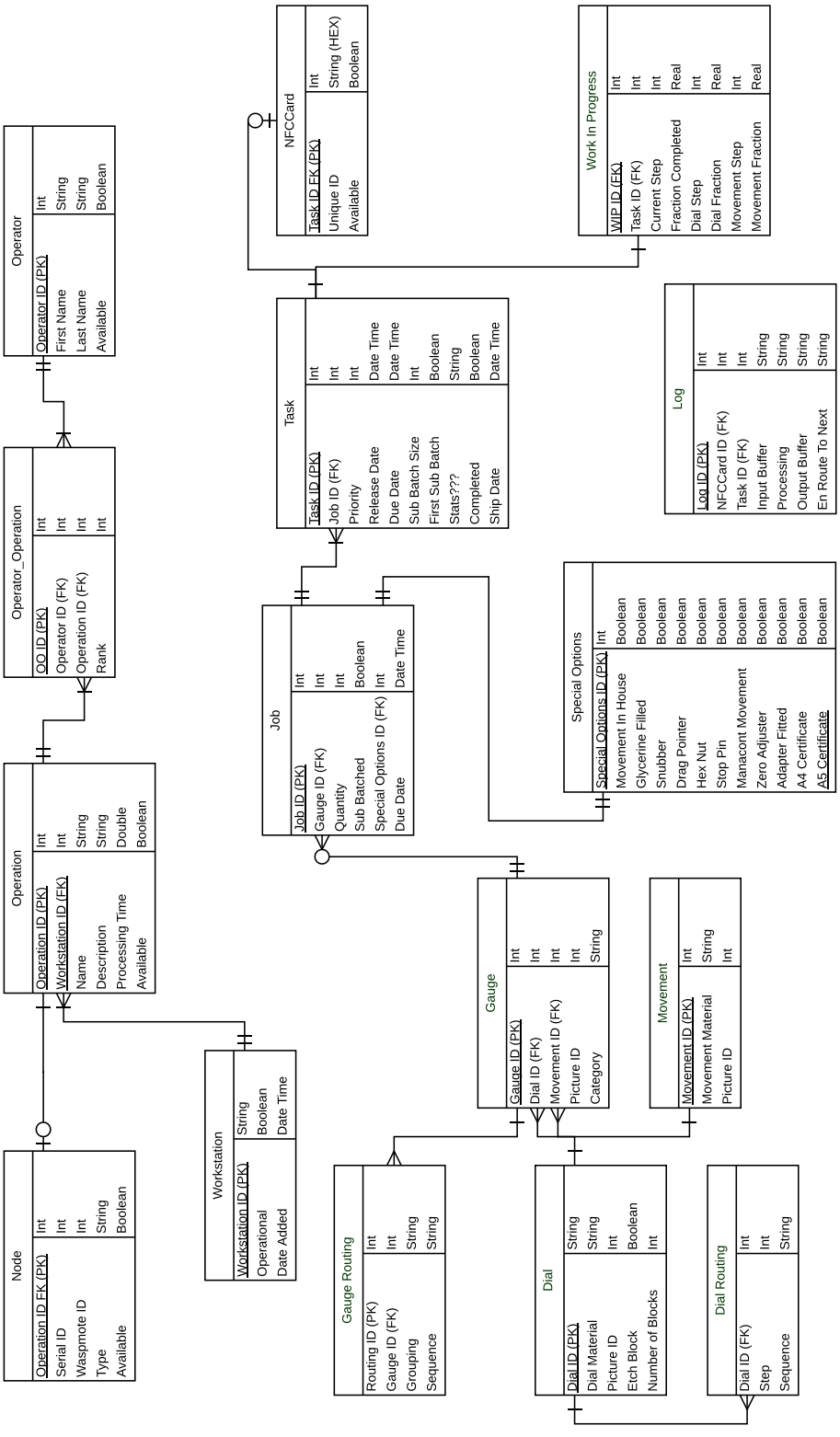


Figure E.17: Entity relationship diagram of the RSA database.

RPS Online Home System Data I/O API About Contact

RFID Cards

Create New

Number	Card Id	Date Added	Available	
1	E26337E9	2014/06/01 8:00:00 AM	☑	Edit | Details | Delete
2	B24C33E9	2014/06/01 8:00:00 AM	☑	Edit | Details | Delete
3	F2092CE9	2014/06/01 8:00:00 AM	☑	Edit | Details | Delete
4	628639E9	2014/06/01 8:00:00 AM	☑	Edit | Details | Delete
5	42222BE9	2014/06/01 8:00:00 AM	☑	Edit | Details | Delete
6	927037E9	2014/06/01 8:00:00 AM	☑	Edit | Details | Delete
7	82C229E9	2014/06/01 8:00:00 AM	☑	Edit | Details | Delete
8	D20E2BE9	2014/06/01 8:00:00 AM	☑	Edit | Details | Delete
9	42C92CE9	2014/06/01 8:00:00 AM	☑	Edit | Details | Delete
10	A22D2EE9	2014/06/01 8:00:00 AM	☑	Edit | Details | Delete

© 2015 - Cameron McOnie - Stellenbosch University

Figure E.18: Management portal interface for managing RFID cards.

Figure E.19: Management portal interface for creating an RFID card.

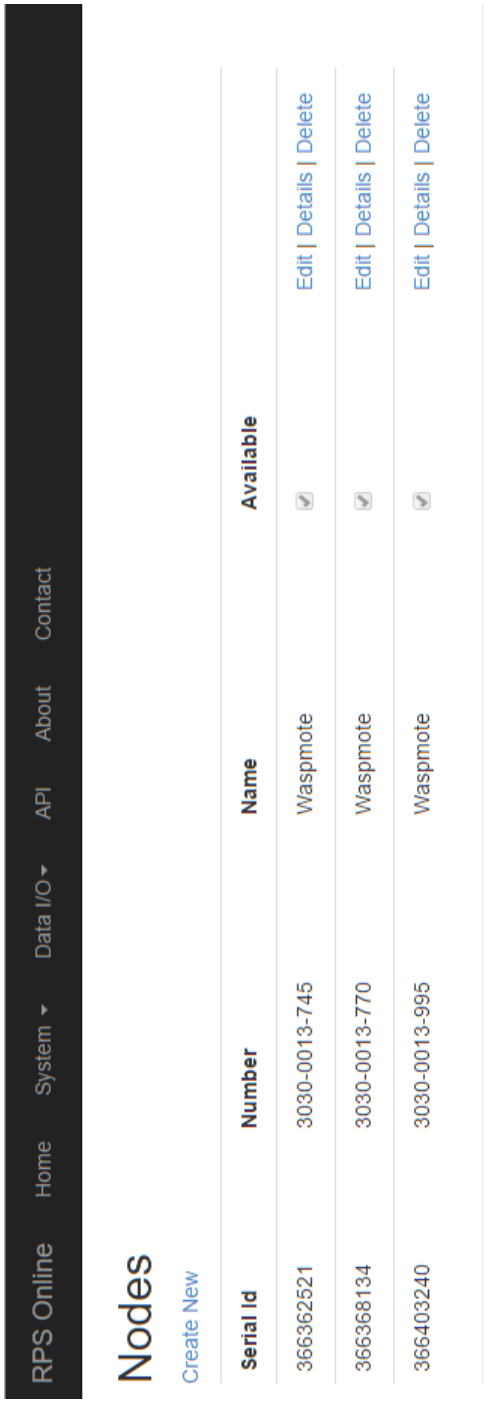


Figure E.20: Management portal interface for managing nodes.

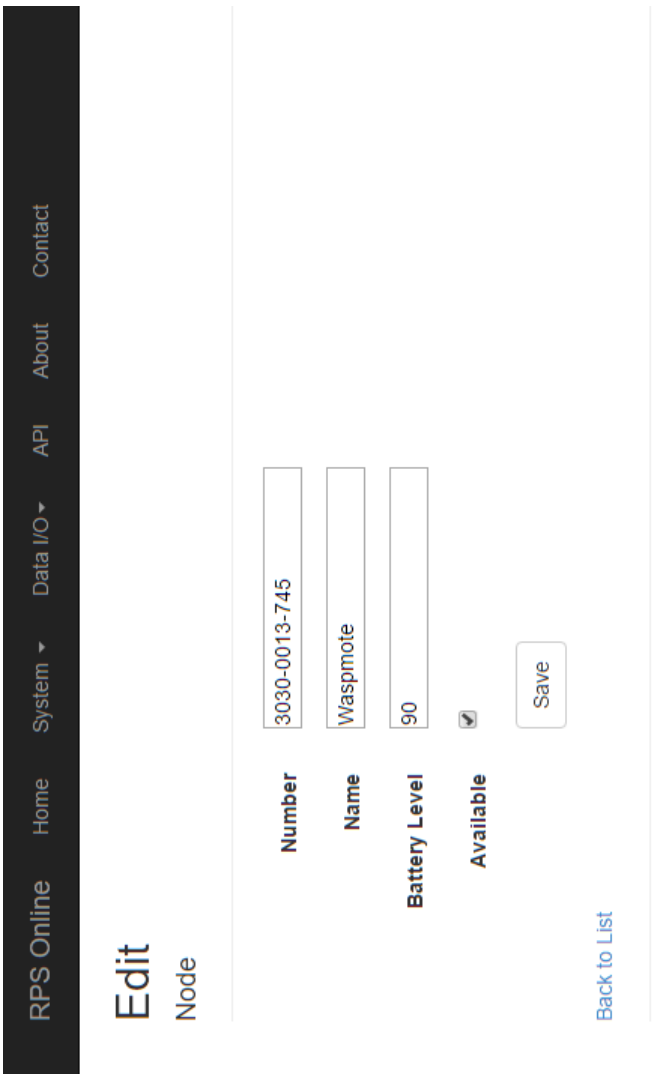


Figure E.21: Management portal interface for editing an RFID card.

Figure E.22: Management portal interface for creating an node.

RPS Online Home System Data I/O API About Contact

Jobs

Create New

Job Id	Gauge Code	Quantity	Priority	Sub Batches	Number Completed	Option Id	Release Date	Due Date	Time Stamp	Status	Expected Completion Time	
2001	PBGA 63	105	10000	3	1	0	2014/10/14 08:00:00	2014/10/16 08:00:00	2014/10/14 15:48:59	WIP		Select | Edit | Details | Delete
2002	PBGA 63	50	10000	1	0	0	2014/10/14 08:00:00	2014/10/16 08:00:00	2014/10/14 15:48:59	WIP		Select | Edit | Details | Delete
2003	PBGA 63	18	10000	1	0	0	2014/10/14 08:00:00	2014/10/16 08:00:00	2014/10/14 15:48:59	WIP		Select | Edit | Details | Delete
2004	PBGA 63	45	10000	1	0	0	2014/10/14 08:00:00	2014/10/16 08:00:00	2014/10/14 15:48:59	WIP		Select | Edit | Details | Delete
2005	PBGA 63	70	10000	1	0	0	2014/10/14 08:00:00	2014/10/16 08:00:00	2014/10/14 15:48:59	WIP		Select | Edit | Details | Delete
2006	PBGA 63	24	10000	1	0	0	2014/10/14 08:00:00	2014/10/16 08:00:00	2014/10/14 15:48:59	WIP		Select | Edit | Details | Delete
3000	PBGA 63	135	10000	3	0	0	2014/10/14 08:00:00	2014/10/16 08:00:00	2014/10/14 15:48:59	NEW		Select | Edit | Details | Delete
3001	PBGA 63	87	10000	2	0	0	2014/10/14 08:00:00	2014/10/16 08:00:00	2014/10/14 15:48:59	NEW		Select | Edit | Details | Delete
3002	PBMA 42	63	10000	2	0	0	2014/10/14 08:00:00	2014/10/16 08:00:00	2014/10/14 15:48:59	NEW		Select | Edit | Details | Delete

Figure E.23: Management portal interface for managing jobs.

Appendix F

Web service API

This appendix provides the application programming interface (API) definitions of the resources exposed off the reactive scheduling application (RSA). Each endpoint was detailed to provide the integrating parties, i.e., mobile clients and sensor networks, with a set of standardised interfaces off of which to develop. Each REST endpoint contains example JSON request and response objects where appropriate. Note, header and path parameters, as well as JSON body properties, are only described once per resource—except if their meaning is different within the context of a particular API call.

F.1 Job

This section describes the endpoints exposed on the **job** resource.

Create a job

Description	Creates a job. Task associations must be specified including the unique identifier of a RFID card. This method allows clients to create a new job when an order arrives. Clients should specify all job information, including information for each task.
URL Endpoint	`http://<#URL#>/api/job/<Job-Id>`
Method	POST
Headers	Content-Type: application/json Accept: application/json
Parameters	**gaugeId** *required* Unique Id of the gauge to be assembled. **quantity** *required* Number of gauges to be assembled. **priority** *required* Priority of the job (defualt = 10 000). Jobs with a priority of less than 10 000 are given preference. **subBatches** *required* Number of tasks that will be associated with this job. **numberCompleted** *required* Number of subtasks completed, set to zero when creating a job. **releaseDate** *required* Date-time at which the job will be released onto the shop floor. **dueDate** *required* Date-time the job is due for dispatch. **subBatchSize** *required* Quantity of gauges allocated to a task. **status** *required* Current state of the task. Set as "NEW" when creating a job. **rank** *required* Dynamic rank allocation. Set to zero when creating a job. **cardNumber** *optional* Identification number of the RFID card to pair with the task.
Response	201 (Created) Returns the location and a representation of the job resource—including all associated tasks.

JSON Request

```
{
  "jobId": 2001,
  "gaugeId": 10,
  "quantity": 100,
  "priority": "10000",
  "subBatches": 2,
  "numberCompleted": 0,
  "optionId": 0,
  "releaseDate": "2014-10-14T08:00:00",
  "dueDate": "2014-10-16T08:00:00",
  "taskks": [
    {
      "taskkId": 200101,
      "jobId": 2001,
      "subBatchSize": 50,
      "status": "NEW",
      "cardNumber": 12,
      "rank": 0
    },
    {
      "taskkId": 200102,
      "jobId": 2001,
      "subBatchSize": 50,
      "status": "NEW",
      "cardNumber": 7,
      "rank": 0
    }
  ]
}
```

Fetch a job

DESCRIPTION Fetches a job resource using the job Id key. All associated tasks are returned in the body. This method is intended for clients interested in accessing job information (including job expected completion times).

URL ENDPOINT `http://<#URL#>/api/job/<Job-Id>`

METHOD GET

HEADERS Content-Type: application/json
Accept: application/json

PARAMETERS **instantiationDate** Date-time at which the job will be released onto the virtual factory floor (simulation model). Note, instantiation date is different to release date—release date is static, instantiation date dynamical shifts with the rolling horizon.

timeStamp Date-time at which the job was added to the system.

ECT Expected completion time of the job. Statistical estimate of the job's predicted completion time based on the best scenario.

status Current state of the task. Possible states are "NEW", "WIP", and "COMPLETED".

rank Dynamic rank allocation. Indicates the task's ranking within its current queue.

inputQueueAssociation Operation Id of any input queue associations the task is engaged in. Note, the task may only reside in one queue at any one time. If the task does not reside in any input queues `null` is returned.

processingQueueAssociation Operation Id of any processing activity associations the task is engaged in. Note, the task may only reside in one queue at any one time. If the task is not engaged in a processing activity `null` is returned.

outputQueueAssociation Operation Id of any output queue associations the task is engaged in. Note, the task may only reside in one queue at any one time. If the task does not reside in any output queues `null` is returned.

dispatchQueueAssociation Operation Id of any dispatch activity associations the task is engaged in. Note, the task may only reside in one queue at any one time. If the task is not engaged in a dispatch activity `null` is returned.

RESPONSE 200 (OK) Returns a job resource along with the associated tasks.

JSON RESPONSE


```
{
  "jobId": 2001,
  "gaugeId": 10,
  "quantity": 100,
  "priority": "10000",
  "subBatches": 2,
  "numberCompleted": 0,
  "optionId": 0,
  "releaseDate": "2014-10-14T08:00:00",
  "dueDate": "2014-10-16T08:00:00",
  "insantiationDate": "2014-10-14T08:00:00",
  "timeStamp": "2014-10-14T15:48:59.247",
  "ECT": "2014-10-15T14:53:00",
  "status": "WIP",
  "OS_Cj": "osJob_2001",
  "matrixExpression": "svJobMatrix[3,2]",
  "matrixRow": 3,
  "taskks": [
    {
      "dispatchQueueAssociation": null,
      "inputQueueAssociation": 101,
      "outputQueueAssociation": null,
      "processingQueueAssociation": null,
      "taskkId": 200101,
      "jobId": 2001,
      "subBatchSize": 50,
      "status": "NEW",
      "rank": 1
    },
    {
      "dispatchQueueAssociation": null,
      "inputQueueAssociation": 123,
      "outputQueueAssociation": null,
      "processingQueueAssociation": null,
      "taskkId": 200102,
      "jobId": 2001,
      "subBatchSize": 50,
      "status": "WIP",
      "rank": 2
    }
  ]
```

```
42 }
```

Fetch all jobs

Description Fetches all job resources. All associated tasks are returned in the body. This method is intended for clients interested in accessing all job related information.

URL Endpoint `http://<#URL#>/api/jobs/tasks`

Method GET

Headers Content-Type: application/json
Accept: application/json

Parameters **timeStamp** Date-time at which the job was added to the system.

ECT Expected completion time of the job. Statistical estimate of the job's predicted completion time based on the best scenario.

status Current state of the task. Possible states include "NEW", "WIP", and "COMPLETED".

rank Dynamic rank allocation. Indicates the task's ranking within its current queue.

Response 200 (OK) Returns a collection of job resources along with their associated tasks.

JSON Response

```
1  [
2    {
3      "jobId": 2003,
4      "gaugeCode": "PBGA 63",
5      "quantity": 18,
6      "priority": "10000",
7      "subBatches": 1,
8      "optionId": 0,
9      "releaseDate": "2014-10-14T08:00:00",
10     "dueDate": "2014-10-16T08:00:00",
11     "insantiationDate": "0001-01-01T00:00:00",
12     "numberCompleted": 0,
13     "timeStamp": "2014-10-14T15:48:59.247",
```

```
    "status": "WIP",
    "expectedCompletionTime": null,
    "OS_Cj": "osJob_2003",
    "matrixExpression": "svJobMatrix[5,2]",
    "taskks": [
      {
        "taskkId": 200301,
        "subBatchSize": 18,
        "status": "WIP",
        "rank": 0,
        "cardNumber": 1
      }
    ]
  },
  {
    "jobId": 2004,
    "gaugeCode": "PBGA 63",
    "quantity": 45,
    "priority": "10000",
    "subBatches": 1,
    "optionId": 0,
    "releaseDate": "2014-10-14T08:00:00",
    "dueDate": "2014-10-16T08:00:00",
    "insantiationDate": "0001-01-01T00:00:00",
    "numberCompleted": 0,
    "timeStamp": "2014-10-14T15:48:59.247",
    "status": "WIP",
    "expectedCompletionTime": null,
    "OS_Cj": "osJob_2004",
    "matrixExpression": "svJobMatrix[6,2]",
    "taskks": [
      {
        "taskkId": 200401,
        "subBatchSize": 45,
        "status": "WIP",
        "rank": 1,
        "cardNumber": 0
      }
    ]
  },
  {
    "jobId": 3001,
    "gaugeCode": "PBGA 63",
    "quantity": 87,
    "priority": "10000",
```

```
    "subBatches": 2,
    "optionId": 0,
    "releaseDate": "2014-10-14T08:00:00",
    "dueDate": "2014-10-16T08:00:00",
    "insantiationDate": "0001-01-01T00:00:00",
    "numberCompleted": 0,
    "timeStamp": "2014-10-14T15:48:59.247",
    "status": "NEW",
    "expectedCompletionTime": null,
    "OS_Cj": "osJob_3001",
    "matrixExpression": "svJobMatrix[10,2]",
    "taskks": [
      {
        "taskkId": 300101,
        "subBatchSize": 50,
        "status": "NEW",
        "rank": 0,
        "cardNumber": 0
      },
      {
        "taskkId": 300102,
        "subBatchSize": 50,
        "status": "NEW",
        "rank": 0,
        "cardNumber": 0
      }
    ]
  }
]
```

Update job priority

DESCRIPTION Allows the priority value to be updated on a job resource. Preference is given to the job during dynamic rank allocation. Note, the entire resource representation is sent—as per rest convention.

URL ENDPOINT `http://<#URL#>/api/job/<job-Id>/ priority`

METHOD PUT

HEADERS Content-Type: application/json
Accept: application/json

PARAMETERS **priority** *required* Numerical priority value to be assigned to the job.

RESPONSE 200 (OK) Returns a representation of the job resource including the modified priority value.

JSON REQUEST

```
1  {
2    "jobId": 2001,
3    "gaugeId": 10,
4    "quantity": 100,
5    "priority": "10",
6    "subBatches": 2,
7    "numberCompleted": 0,
8    "optionId": 0,
9    "releaseDate": "2014-10-14T08:00:00",
10   "dueDate": "2014-10-16T08:00:00",
11   "insantiationDate": "2014-10-14T08:00:00",
12   "timeStamp": "2014-10-14T15:48:59.247",
13   "ECT": "2014-10-15T14:53:00",
14   "status": "WIP",
15   "OS_Cj": "osJob_2001",
16   "matrixExpression": "svJobMatrix[3,2]",
17   "matrixRow": 3
18 }
```

F.2 Task

This section describes the endpoints exposed on the **task** resource.

Fetch all tasks (including queue positions)

Description	Retrieves all tasks, including queue associations. Clients seeking to locate the position of WIP item on the shop floor should retrieve this resource.
URL Endpoint	`http://<#URL#>/api/tasks/queues`
Method	GET
Headers	Content-Type: application/json Accept: application/json
Parameters	**taskId** Unique identifier of the task. **operationId** Unique identifier of the operation at which the task resides. **operationName** Name of the operation at which the task currently resides.. **queueName** Name of the queue in which the task resides.
Response	200 (OK) Returns a collection of tasks including operation data.

JSON Request

```
1  [
2    {
3      "taskkId": 200101,
4      "jobId": 2001,
5      "subBatchSize": 50,
6      "status": "WIP",
7      "rank": null,
8      "operationId": 111,
9      "operationName": "Soldering_BTE",
10     "queueName": "Input"
11   },
12   {
13     "taskkId": 200102,
14     "jobId": 2001,
15     "subBatchSize": 50,
```

```
    "status": "WIP",
    "rank": 3,
    "operationId": 111,
    "operationName": "Soldering_BTE",
    "queueName": "Input"
  },
  {
    "taskkId": 200301,
    "jobId": 2003,
    "subBatchSize": 40,
    "status": "WIP",
    "rank": null,
    "operationId": 112,
    "operationName": "Welding_BTE",
    "queueName": "Processing"
  },
  {
    "taskkId": 200401,
    "jobId": 2004,
    "subBatchSize": 25,
    "status": "WIP",
    "rank": 2,
    "operationId": 112,
    "operationName": "Welding_BTE",
    "queueName": "Dispatch"
  }
]
```

F.3 WIP

This section describes the endpoints exposed on the **WIP** resource.

Description	Receives WIP transaction events from nodes. Parameters are specified in the URL query string. Nodes seeking to upload a transaction event should do so via the URL query string parameters.
URL Endpoint	`http://<#URL#>/api/wip/event?a=<Card Id>&b=<Node Id>&c=<State>`
Method	GET [1]
Path Parameters	**a** *required* RFID card serial Id.
	b *required* Waspmote serial Id.
	c *required* State transition, e.g., "Received".
Headers	Content-Type: application/text Accept: application/text
Response	200 (OK)

F.4 Operation

This section describes the endpoints exposed on the **operation** resource.

Fetch all operations (including queues)

Description	Returns all operations and a collection of tasks for each of the operation's queues and activities. Clients seeking visibility into all operations should retrieve this resource.
URL Endpoint	`http://<#URL#>/api/operations/queues`
Method	GET
Headers	Content-Type: application/json Accept: application/json
Parameters	**operationId** Unique identifier of the operation. **name** Name of the operation. **description** Short description of the operation. **available** Boolean indicating whether the operation is available, i.e. on-line. **inputQueue** Collection containing all the tasks residing in the operation's input buffer. **processingQueue** Collection containing all the tasks currently being processed at the workstation. **outputQueue** Collection containing all the tasks residing in the operation's output buffer. **dispatchQueue** Collection containing all the tasks en route to the next operation. **taskId** Unique identifier of the task. **rank** Dynamic rank allocation. Numeric value indicating the relative priority of the task. **cardNumber** Visible unique identifier.
Response	200 (OK) Returns a collection of operations and a collection of tasks for each associated queue.

JSON Response

```
1 [
2   {
```

```
3     "operationId": 111,
4     "name": "Soldering_BTE",
5     "description": "Soldering of block, tube, and end-
          piece.",
6     "processingTime": "Random.Uniform(0.1,0.3)",
7     "available": true,
8     "timeToRepair": null,
9     "inputQueue": [
10      {
11        "taskkId": 200301,
12        "subBatchSize": 18,
13        "status": "WIP",
14        "rank": 1,
15        "cardNumber": "E26337E9"
16      },
17      {
18        "taskkId": 200301,
19        "subBatchSize": 18,
20        "status": "WIP",
21        "rank": 2,
22        "cardNumber": "B24C33E9"
23      }
24    ],
25    "processingQueue": [
26      {
27        "taskkId": 200201,
28        "subBatchSize": 50,
29        "status": "WIP",
30        "rank": null,
31        "cardNumber": "F2092CE9"
32      }
33    ],
34    "OutputQueue": [
35      {
36        "taskkId": 200102,
37        "subBatchSize": 35,
38        "status": "WIP",
39        "rank": null,
40        "cardNumber": "42222BE9"
41      }
42    ],
43    "dispatchQueue": [
44      {
45        "taskkId": 200401,
46        "subBatchSize": 45,
```

```
        "status": "WIP",
        "rank": null,
        "cardNumber": "A45GH823"
      }
    ]
  },
  {
    "operationId": 112,
    "name": "Welding_BTE",
    "description": "Welding of block, tube, and end-
       piece.",
    "processingTime": "Random.Uniform(0.1,0.3)",
    "available": true,
    "timeToRepair": null,
    "inputQueue": [
      {
        "taskkId": 200701,
        "subBatchSize": 46,
        "status": "WIP",
        "rank": 1,
        "cardNumber": "450923B2"
      }
    ],
    "processingQueue": [],
    "outputQueue": [
      {
        "taskkId": 200801,
        "subBatchSize": 80,
        "status": "WIP",
        "rank": null,
        "cardNumber": "87A07721"
      }
    ],
    "dispatchQueue": []
  }
]
```

Fetch an operation (including queues)

DESCRIPTION Returns an operation and the collection of tasks for each of the operation's queues and activities. Clients seeking to communication task rankings to operators should retrieve this resource.

URL ENDPOINT `http://<#URL#>/api/operation/<operation-Id>/queues`

METHOD GET

HEADERS Content-Type: application/json
Accept: application/json

PARAMETERS **operationId** Unique identifier of the operation.

RESPONSE 200 (OK) Returns an operation and a collection of task for each associated queue.

JSON RESPONSE

```
{
  "operationId": 111,
  "name": "Soldering_BTE",
  "description": "Soldering of block, tube, and end-
      piece.",
  "processingTime": "Random.Uniform(0.1,0.3)",
  "available": true,
  "timeToRepair": null,
  "inputQueue": [
    {
      "taskkId": 200301,
      "subBatchSize": 18,
      "status": "WIP",
      "rank": 2,
      "cardNumber": "E26337E9"
    },
    {
      "taskkId": 200301,
      "subBatchSize": 18,
      "status": "WIP",
      "rank": 1,
      "cardNumber": "B24C33E9"
    }
  ],
  "processingQueue": [
```

```
    {
      "taskkId": 200201,
      "subBatchSize": 50,
      "status": "WIP",
      "rank": 0,
      "cardNumber": "F2092CE9"
    }
  ],
  "outputQueue": [
    {
      "taskkId": 200102,
      "subBatchSize": 35,
      "status": "WIP",
      "rank": null,
      "cardNumber": "42222BE9"
    }
  ],
  "dispatchQueue": [
    {
      "taskkId": 200401,
      "subBatchSize": 45,
      "status": "WIP",
      "rank": 0,
      "cardNumber": "45E9R091"
    }
  ]
}
```

F.5 RFID card

This section description of the endpoints expose on the **card** resource.

Fetch all RFID cards

Description	Returns the collection of RFID cards that are circulating in the RFID card pool. Clients seeking to view a list of all RFID cards should retrieve this resource.
URL Endpoint	`http://<#URL#>/api/rfidcards`
Method	GET
Headers	Content-Type: application/json Accept: application/json
Parameters	**cardId** Internal serial identifier of the RFDI card. **number** Unique numeric visual identifier of the RFID card. **dateAdded** Date-time at which the card was added to the system. **available** Boolean representing the availability of the card. `True` indicates that the card is on-line. **active** Boolean representing the relationship status of the card. `True` indicates that the card is currently paired to a WIP item.
Response	200 (OK) Returns a collection of RFID cards and metadata regarding their associations.

JSON Response

```
1  [
2    {
3      "cardId": "E26337E9",
4      "number": 1,
5      "dateAdded": "2014-10-14T08:00:00",
6      "available": true,
7      "active": true
8    },
9    {
10     "cardId": "B24C33E9",
11     "number": 2,
```

```
    "dateAdded": "2014-10-14T08:00:00",
    "available": true,
    "active": true
  },
  {
    "cardId": "F2092CE9",
    "number": 3,
    "dateAdded": "2014-10-14T08:00:00",
    "available": false,
    "active": false
  },
  {
    "cardId": "42222BE9",
    "number": 4,
    "dateAdded": "2014-10-14T08:00:00",
    "available": true,
    "active": false
  }
]
```

Fetch an RFID card

DESCRIPTION Returns an RFID card. Clients seeking to view the properties of a single RFID card should request this resource.

URL ENDPOINT `http://<#URL#>/api/rfidcards/<rfidcard-Id>`

METHOD GET

HEADERS Content-Type: application/json
Accept: application/json

PARAMETERS **cardId** Internal serial identifier of the RFID card.

number Unique numeric visual identifier of the RFID card.

dateAdded Date-time at which the card was added to the system.

available Boolean representing the availability of the card. `True` indicates that the card is on-line.

active Boolean representing the relationship status of the card. `True` indicates that the card is currently paired to a WIP item, and therefore unavailable for association.

RESPONSE 200 (OK) Returns the location and representation of the RFID card resource.

JSON RESPONSE

```
1 {
2   "cardId": "E26337E9",
3   "number": 1,
4   "dateAdded": "2014-10-14T08:00:00",
5   "available": true,
6   "active": true
7 }
```

F.6 SOE

This section describes the endpoints exposed on the **SOE** resource.

Manual invocation of the simulation optimisation engine

DESCRIPTION Clients seeking to manually invoke an optimisation call should POST to the SOE resource. A simulation experiment will automatically execute, task ranking will be refreshed, and the results returned.

URL ENDPOINT `http://<#URL#>/api/soe`

METHOD POST [1]

HEADERS Content-Type: application/json
Accept: application/json

PARAMETERS **previousRun** Date-time of the previous SOE optimisation run.

runCount SOE cumulative run counter.

bestScenario Selected scenario from the previous SOE run, e.g., SPT.

objective Objective function of the SOE.

RESPONSE 201 (Created) Returns the location and representation of the SOE resource.

JSON RESPONSE

```
1 {
2   "previousRun": "2014-11-18T02:59:01",
3   "runCount": 63,
4   "bestScenario": "SPT",
5   "objective": "Makespan"
6 }
```

[1] POST is used as the server will experience a state change after each request.

Appendix G

Component interfaces

G.1 Simulation Optimisation Engine

Simio File IO

Name	File IO Interface.
Type	Reactive scheduling application component.
Description	The SimioFileIO class is responsible for handling Simio project file input and output. This class retrieves a .spfx file from the specifed location on disk, loads the file into memory, and returns an ISimioProject to the caller.
Public	**loadProject** Loads a Simio project file from disk. File name is handled internally.

Experiment Runner

NAME	Experiment Interface.
TYPE	Reactive scheduling application component.
DESCRIPTION	This class provides a programmatic interface for objects seeking to execute an experiment. A method to create a CMExperiment is provided which is usefull for extracting experiment results (Piviot Grid information from within Simio—not available on the standard IExperiment).
PUBLIC	**runExperiment** *IExperiment* Runs a Simio experiment.
	runExperimentAndCreateCMExperiment *IExperiment* Runs a Simio experiment and creates a CMExperiment.

Scenario Comparator

NAME	Scenario Comparator Interface.
TYPE	Reactive scheduling application component.
DESCRIPTION	The Scenario Comparator class declares a programmatic interface to objects interested in selecting a particular, random, or the best[1] scenario from an executed experiment.
PUBLIC	**selectByName** *name, IExperiment* Selects the scenario according to a specified name from the set of candidate scenarios.
	selectByRandom *IExperiment* Selects a random scenario from the set of candidate scenarios.
	selectByKimNelson *IExperiment, IZ, CL, Max, NZero, c* Selects the best scenario by using the Kim Nelson ranking and selection procedure of Kim & Nelson (2001).

[1] with respect to a statistical selection method, e.g., Kim Nelson

Experiment Input Analyser

Name	Experiment Input Interface.
Type	Reactive scheduling application component.
Description	The ExperimentInputAnalyser class defines a programmatic interface for objects interested gathering information about the state of the system in order to setup an experiment and synchronise the SOE with the captured state. This class is responsible for preparing the set of data to be fed into the experiment.
Public	**determineExperimentStartingTime** Determines the experiments starting time.
Private	**updateStagnantJobsInstantiationDate** Updates the job's instantiation date. This is necessary since the rescheduling point may have passed the job's release date, preventing the job from being released into the simulation model. E.g. Job A was scheduled to be released at 9:00 AM but, in reality, was not (no entry event received). If the next rescheduling point is set for 9:01 AM, the job may not be overlooked..

Experiment Output Analyser

NAME	Experiment Output Analyser Interface.
TYPE	Reactive scheduling application component.
DESCRIPTION	The ExperimentOutputAnalyser class defines a programmatic interface to objects that intend to update the system's decision support data and meta-data. This includes job's expected completion times, reordering of tasks in input buffers, and system metrics. Objects interested in updating task rankings or system responses should possess an IScenario. Objects interested in updating job expected completion times should possess a CMSceanrio.
PUBLIC	**updateProjectedSystemMetrics** *IScenario* Updates the system's metrics table with a scenario's output statistics.
	updateJobECT *IScenario* Updates the expected completion time of each job which was simulated.
	rankAllOperationsInputBuffers *IScenario* This method ranks each tasks within each operation's input buffer. This rank value is used by operators on the shop floor for decision making.
PRIVATE	**rankBySIRO** Rank the list of tasks in random order.
	rankBySPT Rank the list of tasks in descending order using the processing time as key.
	rankByLNG Rank the list of tasks in descending order using the sub-batch size as key..
	rankByLPT Rank the list of tasks in ascending order using the processing time as key.
	rankByERD Rank the list of tasks in descending order using the release data as key..
	rankByEDD Rank the list of tasks in descending order using the due date as key.

Appendix H

Additional material

Table H.1 shows a summary of the software used for the creation of the reactive scheduling system.

Table H.1: Development tools, languages, libraries, and paradigms used.

Item	Hardware	Language	IDE	Frameworks/Libraries	Paradigms
Sensor network	Waspmote Pro	C/C++	Waspmote IDE	WaspWIFI.h, WaspRFID.h	Procedural
Mobile client	Apple iPad (3rd Gen)	Swift	Xcode	HTTP Networking	Object-orientated, Delegation, Model-view-controller
Reactive scheduling application	Virtual machine	C#	Microsoft Visual Studio	Web API 2.0, ASP MVC 5.0	Object-orientated, Model-view-controller
Simulation	PC	Graphical	Simio®	Standard	Object-orientated

Table H.2: Technical specifications of the computer used for experimentation.

Component	Specification
PC	
Processor:	Intel® Core™ i7-4790K 4.0 GHz
Memory:	8 GB
Hard disk:	256 GB Solid state drive
Graphics card:	NVIDIA GeForce GTX 550 Ti
Operating System:	Windows 10 (64 bit)
Monitor	
Size:	24-inch (16:9)
Resolution:	1920 by 1080 pixels