NTNU – Trondheim
Norwegian University of
Science and Technology

Joint Maintenance Interval and Spare Parts Optimization using a Discrete-Event Simulation Model

Arjen Martens

Reliability, Availability, Maintainability and Safety (RAMS)
Submission date: July 2015
Supervisor: Jørn Vatn, IPK
Co-supervisor: Trond Østerås, IPK

Norwegian University of Science and Technology
Department of Production and Quality Engineering

RAMS

Reliability, Availability,
Maintainability, and Safety

Joint Maintenance Interval and Spare Parts Optimization using a Discrete-Event Simulation Model

Arjen Martens

July 2015

MASTER'S THESIS

Department of Production and Quality Engineering

Norwegian University of Science and Technology

Supervisor 1: Professor Jørn Vatn

Supervisor 2: Trond Østerås

Preface

This report represents the Master's thesis of the Master's programme in Reliability, Availability, Maintenance and Safety (RAMS) of the Norwegian University of Science and Technology (NTNU). The subject is part of the risk-based maintenance optimization specialization track of the RAMS programme.

During this project a Discrete-Event Simulation (DES) in Visual Basic for Applications (VBA) has been developed which is used for joint maintenance interval and spare parts optimization. The case on which this simulation is based, has been developed by the author based on data from Statoil and the OREDA Handbook.

Trondheim, 10-07-2015

Arjen Martens

Acknowledgment

First of all, I would like to thank Jørn Vatn for his guidance throughout this semester. Jørn has helped me with developing the model. Whenever I faced a problem during the programming, he showed the possible solutions.

Furthermore, I would like to thank Trond Østerås for helping me with developing the case. Unfortunately, there were some issues on our way, but I am really thankful that you kept supporting me despite these issues. I am really happy that we were finally able to develop a good case which I could work with.

Lastly, I would like to thank Statoil for making their data available to me.

A.M.

Summary and Conclusions

The goal of this report is to use discrete-event simulation (DES) as a method for optimizing maintenance strategies, such as spare parts levels and maintenance intervals. Firstly, the author argues for spare parts optimization with a DES in Visual Basic for Applications (VBA). The models and assumptions that are needed for developing such a model are explained. Furthermore, this report elaborates on how a DES can be coded in VBA. Lastly, several methods for optimizing both speed and decision variables of a DES are introduced.

The report shows how a DES can be coded and which models and assumptions can be used in developing such a simulation. A specific focus is on the design of the pending-event set (PES), which is the core of the DES. Several different designs are tested in different situations in order to determine their performance. The results show that the performance of these methods vary in each situation, and therefore the designer of a DES should determine the characteristics of the DES, before an appropriate PES method can be chosen. This thesis shows that a simplified genetic algorithm can be used in order to find good results in a faster and more structured way than a trial-and-error method. It furthermore shows that this genetic algorithm can be used for joint optimization of preventive maintenance interval, the overhaul interval, spare order threshold and stock levels.

The author concludes the report with recommendations for further work. On the practical side, the impact of different PM strategies on stock levels should be researched. Furthermore, the research to including condition-based maintenance (CBM) in a model like this should be taken a step further with a more complex model for CBM. On the theoretical side, the PES methods should be more thoroughly studied. More functions to manipulate the PES and the required memory space should be included in further research. Lastly, the author believes that the simplified genetic algorithm can be further improved, which can be a focus topic in further research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The relation of preventive maintenance (PM) with inventory costs can seem unclear, since the demand for replaceable parts decreases as the replacement interval increases and is minimum for a failure replacement policy, where items are only replaced upon failure (Barlow and Proschan, 1964). Hence, with a preventive replacement policy one needs more parts, which results in an increase of inventory related costs of these spares again. However, a higher PM frequency leads to a better predictable demand for spare parts and hence to a lower spare parts safety stock (de Smidt-Destombes et al., 2009). The replaceable parts used for the preventive replacement can be delivered according to the just-in-time (JIT) principle, which results in no storage costs for these parts. Van Horenbeek et al. (2013) state in their review paper the importance of joint maintenance interval and inventory optimization. Models that jointly tackle both optimization problems give better optimal solutions, since they do not inherit certain assumptions like most maintenance interval optimizations models have, for example: infinite number of available spare parts, perfect repairs or no lead times for spares. Besides that, they do not take inventory related costs into account, which might drop significantly by just a small increase in other maintenance related costs. The models described in this research paper tackle very basic systems, with only one component that has only two states. They therefore argue that more research should take place on joint optimization by simulating complex systems. Alrabghi et al. (2013) optimize maintenance and spare parts in a multi-component system through a combined discrete-event and continuous simulation.

Condition-based maintenance (CBM) is a maintenance program that recommends maintenance decisions based on the information collected through condition monitoring (Jardine et al., 2006). Online CBM means that monitoring takes place continuously, while offline CBM means that monitoring only takes place after each test interval. Wang et al. (2008) show in their paper how stock levels can be optimized by using CBM. They introduce a spare order threshold, in addition to the preventive replacement threshold in the classical CBM model.

So far models were either focussed on joint optimization of the PM interval and stock levels, on joint optimization of the test interval and stock level or on finding the optimal spare order threshold when condition monitoring takes place. This thesis tries to jointly optimize all four factors by discrete-event simulation (DES), since this has not been done before to the best of the author's knowledge. Simulation on joint optimization of maintenance and spares in multi-echelon supply systems has not been done either, which is also incorporated in this thesis.

1.2 Objectives

This thesis has two practical objectives:

1. Determine the optimal stock values for the Kristin case.
2. Determine the impacts of having different preventive maintenance strategies on stock levels.

Besides those practical objectives, it has several theoretical objectives, which are related to using a DES for a joint maintenance and spares optimization:

1. Determine a fast method for handling the pending-event set in a discrete-event simulation.
2. Determine an accurate and fast method for optimizing the decision variables of a discrete-event simulation.
3. Determine an accurate and fast method for joint optimization of the preventive maintenance interval, the overhaul interval, spare order threshold and stock levels.

1.3 Limitations

The data acquisition for the Kristin case is very limited and results for this case are therefore not really useful for practical purposes. The developed DES tool inherits some assumptions which could give some different results than when other assumptions are made. Results should therefore be tested more extensively. Furthermore, the tool is not extensively verified by an external coder, which means that the accuracy of the tool is not guaranteed.

1.4 Approach

A DES is made as basis for this research. A literature has been conducted before by the author during the specialization project and is therefore not incorporated in this report. However, the information of this study is used in this report. Based on this information an algorithm is developed and tested with the DES. Furthermore, methods of implementing a pending-event set (PES) of a DES are tested here by simulation.

1.5 Structure of the Report

This reports continues with five more chapters. Chapter 2 describes which modeling approaches are used, while chapter 3 elaborates on how these models are used for programming the DES. Chapter 4 proposes optimization methods for both optimizing output variables and speed of the DES. The results of the simulations are shown in chapter 5. Chapter 6 concludes this thesis.

Chapter 2

Modeling Approaches

This chapter elaborates on the models that are used in this thesis. In section 2.1, the author briefly introduces some methods for spare parts optimization and argues why a DES in VBA is chosen. Section 2.2 introduces which models and policies for spare parts, maintenance and so on, are used in the DES.

2.1 Spare Parts Optimization Methods

There are numerous different options for optimizing the amount of spares, of which some are introduced here. This section clarifies the choice for a DES in VBA for this project.

2.1.1 Introduction to Spare Parts Optimization Methods

One widely spread spare parts optimization technique is Markov Models, about which elementary information can be found in Ross (2014). A shortage in spares can be denoted by the number of backorders (BO). In a Markov model this can be modeled as a state with a negative amount of spares in stock. By finding the steady state probabilities one can get the expected number of BO and use this in a cost formula that takes the capital costs for stocking and the cost for unavailability into account, in order to find the optimal amount of spares in stock. The advantage of this technique is that one gets accurate analytical results. However, this technique suffers from the so called state space explosion problem, which occurs when problems get bigger. It is therefore impossible to model most complex systems realistically with Markov models. Since the problem in this thesis is a complex one, this technique is not suitable. Furthermore, joint optimization of spare parts and maintenance intervals seems like a very challenging task with Markov models.

Another technique that can be used is Petri Nets, which is a graphical and mathematical tool that is applicable to information processing systems. An introduction to Petri Nets can be found in Murata (1989). One popular Petri Net simulation tool is Colored-Petri Nets (CPN) Tools, about which more information can be found in Jensen and Kristensen (2009) and van der Aalst and Stahl (2011). The author has good experiences with this tool, but decides not to use this tool because one loses some modeling freedom when using a tool like this. The author expects it would be difficult to make changes to the model, which is also confirmed by Wells (2002). Furthermore, enabled transition are executed in a random order and can only be in control with prioritizing transitions. Westergaard and Verbeek (2011) show this prioritizing can be very extensive and is therefore not always desirable to do. Lastly, when models get complex, the graphical representation of the Petri Nets can become very complex as well.

Discrete-event Simulation (DES) simulates the dynamics of the real world on an event-by-event basis and is one of the mainstream computer-aided decision-making tools (Law, 2007). It utilizes a mathematical/logical model of a physical system that portrays state changes at precise point in simulated time (Nance, 1993). A short introduction to DES can be found in, for example, Robinson (2014). DES can be used when it becomes analytically impossible to analyze the system and simulation is necessary in order to determine the system's performance. The flexibility and possibility to simulate large systems are the main reasons for opting for developing a DES tool for this case. Furthermore, as stated in 1.1, the use of DES for spare parts optimization is not as extensive studied as, for example, Markov models. Therefore, it is also more interesting for this research to develop a DES.

2.1.2 Discrete-Event Simulation Methods

There are three options for developing DES models: spreadsheets, specialist simulation software and programming languages (Robinson, 2014). Spreadsheets require programming constructs, like Visual Basic, to model more complex systems, while programming languages are used when systems get very complex. Most systems, however, can be modeled using specialist simulation software. Arena (2014) is one of these tools and has been used by the author before. The experience dictates that also within these special packages one quickly requires some programming in order to model systems which sufficiently represent reality.

The main reasons for choosing for spreadsheets, supported by VBA, are the limited programming experience of the author and the user-friendliness of spreadsheets. The author is not an expert in programming. Since the Visual Basic programming language has a steep learning curve and the development time is rather short, success is most likely achieved by using Visual Basic as programming language rather than a more complex one. However, more important, the use of spreadsheets is very user friendly. Users can easily enter their own data in the Excel spreadsheet. The tool is therefore not only applicable to the specific case explained in chapter 3, but can be used for similar situations with different input data. When a DES is created in another programming language which is not supported by an easy user-interface like the Excel spreadsheet, it can be more challenging to re-use the tool. It is furthermore easier for the user to create some different situations in this tool, like the possibilities of having emergency transfers or PM and CBM.

2.2 Models, Policies and Assumptions in the DES

The unlimited freedom one has during creating a DES in VBA requires choosing of several models and approaches for modeling the reality as close as possible. The approaches the author takes in this DES are explained in this section. This section forms the basis for the detailed explanation of the DES in chapter 3. The choices that are made here, are based on the situation of the Kristin platform of Statoil. They are made to approximate the situation as close as possible. The Kristin platform is situated on the South-Western part of the Haltenbanken field. The Kristin platform produces 10 million cubic meter gas a day, which is compressed on the platform before transportation. This installation for the compression process consists of a gas compressor, which is supported by a gas turbine. The Kristin platform is designed with low to no redundancy. A failure of the gas turbine therefore results in a shutdown of the platform. In the remainder of the report this situation is referred to as the Kristin case.

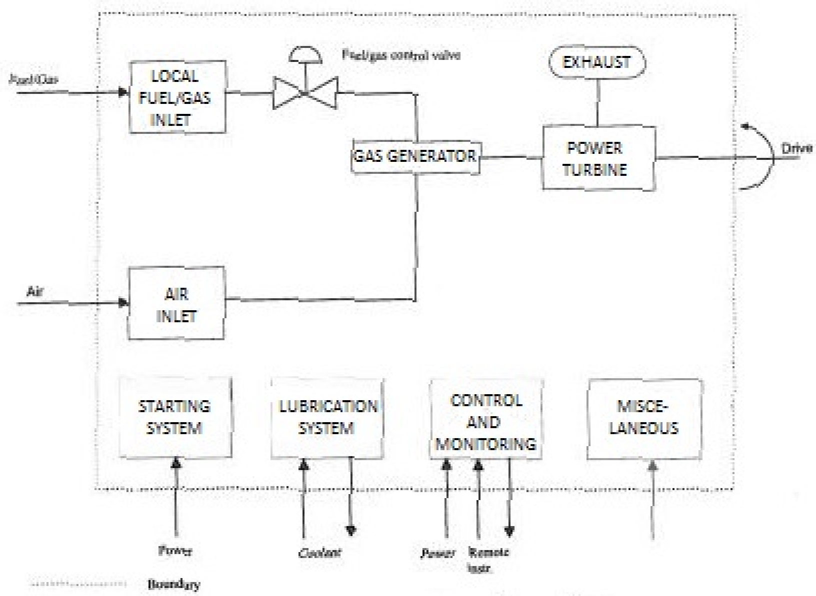

Figure 2.1: Gas Turbine, Boundary Definition (SINTEF, 2009)

2.2.1 Components

The model is based on a gas turbine on the Kristin plant, which is critical to the platform's production. The boundary definition of this gas turbine is given in figure 2.1. Each subdivision of the turbine consists of several maintainable items, such as valves, seals, casings and a control unit. In this model it is assumed that there is no redundancy on these maintainable items (MI). Hence, a failure of one of the MIs leads to a failure of the turbine, which consequently leads to a shutdown of the platform. The model handles only one type of MI per simulation. The amount of spares one need per MI are therefore to be optimized separately. This method makes a code that is much easier to comprehend and still gives accurate results if the availability is high.

2.2.2 Situation Sketch

Spares for the MIs can be stored at the base or the platform. In order to make the case more generic and realistic to reality, it is assumed that there are five identical platforms which are supported by one onshore base, which is a classic two-echelon system. The situation is sketched in figure 2.2. Supply from base to the platforms is either done by boat or in case of an emergency transport by helicopter. It is assumed that these transports can always take place. Lateral ship-

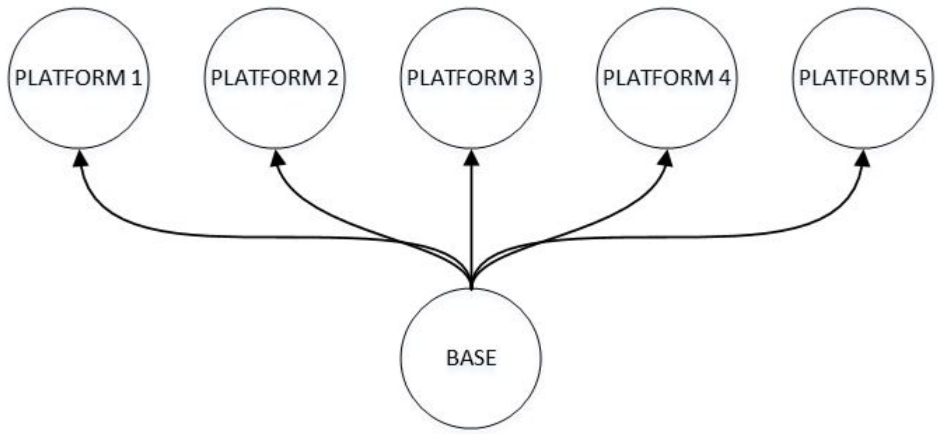

Figure 2.2: Two-Echelon Situation

ments from one platform to another are not possible. The only interaction of this system with the external environment is supplies to the base from an external supplier. Since the platforms are assumed to be independent and identical, the stock policy have the same optimal values for each of the platforms.

2.2.3 Failures

Failures and the states in which the system can be, are represented with a Markov model. The failure rates are exponentially distributed. It is assumed that there is no redundancy, so failure of one component will lead to failure of the platform. SINTEF (2009) uses three different failure types: incipient, degraded and critical. This tool, however, uses only degraded and critical failures. There are therefore three different states for components and platforms: functioning, degraded and failed. Hokstad and Frøvig (1996) state that critical failures can happen due to shock failures or critical degraded failures. The critical failure rate given in SINTEF (2009) does not make this distinction and the failure rate therefore needs some manipulation. Hokstad and Frøvig (1996) show that this can be done by determining the ratio of degraded critical failures and shock critical failures based on the failure mechanisms. Failure mechanisms such as corrosion, fatigue and vibration are classified as degraded, while failure mechanisms like electrical failure, no power and software failure are classified as shock. This gives a degraded-shock ratio. The overall critical failure rate is then assigned to critical shock and critical degraded failure rate according to this ratio. Figure 2.3 shows the Markov model.

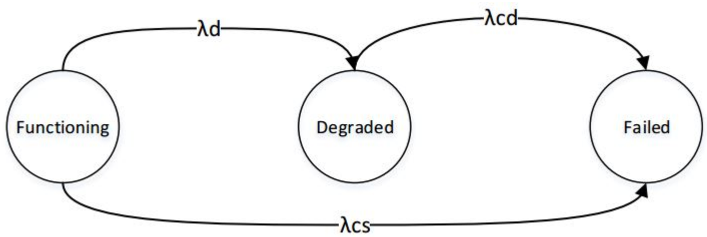

Figure 2.3: States and Rate of the System

Failures are generated upon initialization of the system and upon replacement of the components. The failures are added to the PES and when the clock reaches the failure time the failure is executed. This means that it is possible that two critical failures are close to each other on the timeline and that the second failure occurs when the platform is in a failed state due to the first failure. When the availability is high, this does not cause significant problems for the simulation, since the probability of a failure during downtime of a platform is very low. However, when the availability more failures happen in the simulation than in a real-life situation, which causes an even lower availability. For this reason, results that yield a low availability should be cautiously analyzed.

2.2.4 Stock and Order Policy

In the industry, different kind of inventory policies are used, for which table 2.1 gives an overview. One can classify these under continuous review, where reordering takes place when the stock level reaches s and periodic review, where every time interval R an order is placed. Another classification that can be made is ordering up to a stock level (S) or ordering with a certain batch size (Q). Nowadays inventory policies are usually monitored continuously, as inventory systems are stored in computerized database systems. Policies with a batch size ordering are often used for smaller, less-expensive products, that come in batches of a certain amount Q (for example: boxes of screws, bolts or pens). The other policy is used for items that do not have to ordered in batches (for example: computers, air-conditioning systems). The $(S-1,S)$ policy is a special case of the (s,S) with re-order level $s = S-1$, which is designed for very expensive, slow-moving spares (Sherbrooke, 2008). Since the author wants to have some more flexibility in the type of components that are modeled, the (s,S) policy is chosen, rather than the $(S-1,S)$, which is often used in mathematical models in the literature.

Table 2.1: Overview of Inventory Policies

Notation	Policy
(s,S)	Continuous policy with re-order level s and order-up-to level S
$(S-1,S)$	Special case of (s,S) with re-order level $s = S-1$
(s,Q)	Continuous policy with re-order level s and ordering batch size Q
(R,S)	Periodic policy with re-order interval R and order-up-to level S
(R,s,Q)	Periodic policy with re-order interval R, re-order level s and order-up-to level Q

When an order has been sent out, a new order can only take place after this order is delivered. This holds for orders at both the platform as base. Order times from the base to external suppliers are assumed to be deterministic. This assumption holds in real life when suppliers are reliable and good contracts are signed. However, the order time can in real life depend on the amount of ordered products. This is not taken into account, which means that each order has the same order time, regardless whether the order is for 1 or 100 products. The orders normally have approximately the same size, so the variance on this time due to variance in order size is negligible. Order times from the platform to the base are assumed to be uniform distributed, with as lower level the order handling time and as upper level the sum of this order handling time and the maximum time between two different transfers. In this system it is assumed that the platform is supplied with an exact interval, the time between two different transfers. Orders are made when the stock level at the platform reaches or goes under s, which occurs after a request for a spare part is placed. These requests occur randomly, since the failure rate is exponentially distributed. Therefore it can be assumed that the order is placed randomly in the time interval between two different transfers.

Backorders (B) are created in case the base does not have enough spares to match the order quantity of the platform. The order quantity becomes the amount of spares at stock in base. The difference between these quantities becomes a BO. For example, when a platform orders 5 spares to a base, but the base only has 2 spares, 2 spares are send to the platform and the BO becomes 3 spare parts. BOs from platforms to base are handled with the first-in-first-out (FIFO) policy. In case the stock in base is not sufficient to handle a complete BO of a platform, the difference between the amount of spares requested in the BO and what is delivered to the platform goes back to the queue. This new BO is placed at the back of the queue, this to ensure that platforms get more evenly distributed. For example, when there is a BO of 3 spares to a platform, but there is only one spare in the base stock left, 1 spare is send to the platform and the other 2 spares are send to the back of the queue.

2.2.5 Maintenance Policies

When a component has to be replaced due to a degraded or critical failure, a spare part is taken out of stock and is replaced with a new item. It is assumed that there is an unlimited repair capacity for the replacements at the platform and repairs at the base.

The model comprises of several PM strategies, which can be turned on or switched off by the user of the tool. During PM all components are tested and degraded components are being replaced. This is thus a form of offline CBM. This condition testing does not only take place during a scheduled PM period, but also upon a critical failure. During the shutdown of the platform scheduled preventive replacements are executed when there are enough spares in stock and when the component is in a degraded state. The PM periods furthermore follow the age-replacement policy (ARP), which means that when PM is executed after a critical failure, the next interval is rescheduled and takes place one PM interval after the critical failure. The PM period that was already planned is cancelled. Besides this offline CBM, the model has the option of online CBM, which is assumed to detect any degraded failure immediately when this failure occurs. When the platform has online CBM a PM order is send out for the next PM period upon occurrence of the degraded failure and the necessary spare can be reserved. Lastly, the model has the option of overhauls. During an overhaul all components are replaced with new components and the old ones are discarded. Overhauls follow a block-replacement policy (BRP) and take place each overhaul interval, which is significantly larger than the PM interval. Overhauls do not influence the amount of spares in stock. This assumption can be made because overhauls are planned a long time in advance and the new components can therefore be delivered according to the just-in-time (JIT) principle.

2.2.6 Simulation length

The simulation length depends on both the number of runs (n) and length of one run. As the length of one run is equal to the design life of the platform, only the number of runs has to be determined. Winston (2000) states that the required number of runs (n) can be calculated by using equation 2.1. The author executes 100 trial runs in order to determine the average and estimated standard deviation (SD) of the output. One can then chose the desired margin of error (E) and the desired confidence interval α. Choosing a good simulation length is important in order to find the right balance between accuracy of the results and computation time of running the model.

$$n = \left(\frac{Z_{\alpha/2} \cdot SD}{E}\right)^2 \tag{2.1}$$

Chapter 3

Discrete-Event Simulation for Spare Parts Optimization

This chapter gives a detailed introduction to the DES. Section 3.1 shortly describes the interface of the tool that is created. The remainder of this chapter goes more into the details of the DES. Section 3.2 describes the logic of the DES in relation to the PES, section 3.3 elaborates on the code and section 3.4 gives an overview of the quantities that are used in the DES.

3.1 Interface of Discrete-Event Simulation Tool

The DES tool is programmed in VBA, because it has as main advantage that it is easy for any user of the tool to enter their own data in the Excel-file and run the simulation. The user-interface is shown in Figure 3.1. The file is protected against wrong input of the user, for example in the PM field only the values *"TRUE"* and *"FALSE"* are possible and only positive values are accepted for the and failure and repair rates. This secures the integrity of the simulation. Since the input can be changed by the user, this tool can be used for different cases. However, these cases should follow the same models, policies and assumptions that section 2.2 presents.

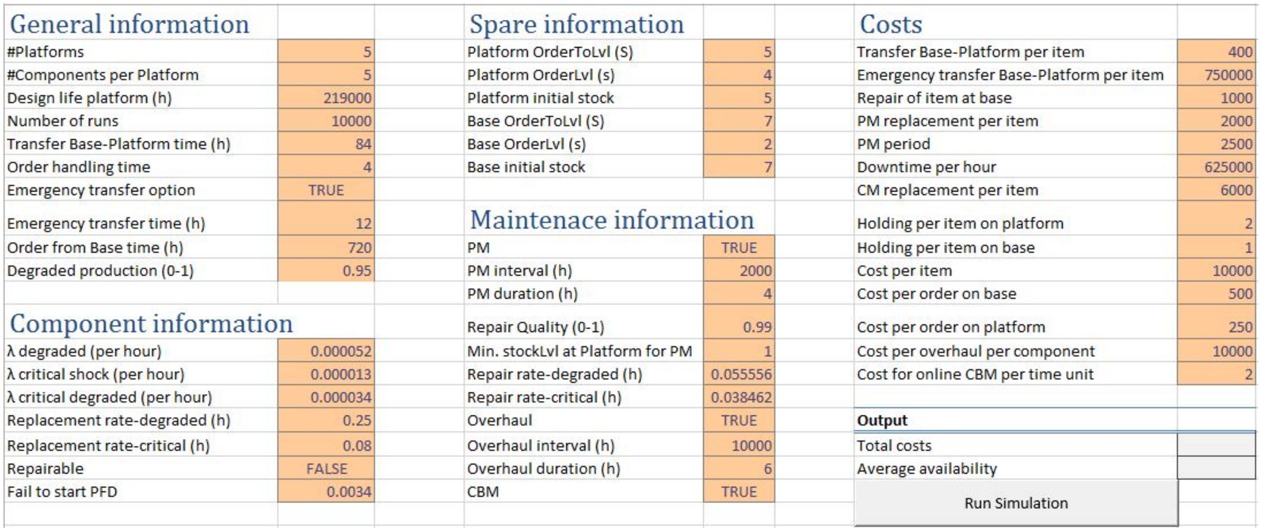

Figure 3.1: User-Interface of the Simulation Tool

3.2 Logic of DES

This section explains the logic of the DES and which functions are executed on the PES in order to create a better understanding of the functionality of the PES. Once a better understanding of the required functionality of the PES is created, a more efficient method of storing the PES and search algorithms can be implemented in the DES. It is therefore essential to document the required functions for the PES before developing the DES.

Each item in the PES consists of a timestamp, function name and optionally a component or platform number and order quantity or failure criticality. Table 3.1 shows which attributes each item in the PES has. The two most essential functions for handling these items in the PES are *deleting the next event* and *adding an event*. *Deleting the next event* takes place when all the actions of the previous event are carried out and the next event has to be retrieved. The event that has the lowest timestamp is taken out of the list and the corresponding function is executed. In a linked list situation the next event is the first event of the list and can therefore be quickly retrieved. Deletion of this event easily takes place by setting the pointer of the list head to the pointer of this first event. In a tree structure, like method 4 in section 4.3 the next event is usually not the first event and it takes therefore several steps reaching this event and hence deleting it. Retrieving the first event in a tree structure takes therefore more time than in a normal linked list.

Deleting an event does not only take place when the next event has to be called, but it also takes place when, for example, a critical failure occurs and PM takes place upon this failure. The "OnStartPMPeriod" and "OnEndPMPeriod", which are already in the PES for the corresponding platform, have to be deleted from the list, as PM does not take place at these times any longer. Furthermore, if PM actions are already scheduled for components at this platform and there are enough spares available to execute these in the downtime of the critical failure, these PM actions are to be deleted from the list as well. Furthermore, reserved transfers of spare parts for these PM actions are to be deleted from the list as well, as these PM actions do not take place at the original time any longer. This search only takes place until the next PM time, and therefore, in the Kristin case, only searches through the first part of the list. On the contrary, when an overhaul takes place, a search throughout the whole list takes place. The search checks every event and deletes the events accordingly.

Table 3.1: PES Items with Corresponding Attributes

Function	Attribute 1	Attribute 2
OnComponentFailure	Component Number	Failure criticality
OnComponentReplacement	Component Number	
OnSpareRepair		
OnPlatformOrderArrival	Platform Number	Quantity
OnBaseOrderArrival		Quantity
OnPMReplacement	Component Number	
OnStartPMPeriod	Platform Number	
OnEndPMPeriod	Platform Number	
OnStartOverhaul		
OnEndOverhaul		
OnPlatformEmergencyArrival	Platform Number	
OnPlatformOrderArrivalSpareReserved	Platform Number	Quantity

Adding an event takes place during the initialization of the DES. During the initialization of components, for example, the event "failure" is inserted in the list for each component. During the executing of an event it may be necessary to add one or more events to the PES. For example, upon a critical failure a corrective maintenance action is added to the PES, and, additionally, a new spare order on the platform might be added to the PES. The author discovers, after observing many trial simulations, that a majority of the events are added in the beginning of the PES, which means in the first 20% of the list. This is caused by a very short handling time for many of the events, after the initialization of the DES. For example, the delivery times or repair times are relatively short in comparison with the failure times. While working with an indexed list, like method 2 in section 4.3 once can use this fact for choosing to add some of these events without looking in the indexed, but immediately in the linked list itself. If the probability is very high that it falls in the first segment of the list, it could save some time that the indexed list is not being called.

Besides these functions that manipulate the PES, there are also several assisting functions. The most important one is the *clock*. This one keeps track of the time of the simulation and is changed every time a new event is called from the list. The *time elapsed* function calculates the difference between the new time of the clock and previous one, and is used for several calculations, for example for the calculation of the downtime costs. The time elapsed function is furthermore used for *calculating the uptime* of each platform. *Calculation of the average availability* uses these uptimes upon termination of the simulation in order to determine this output variable.

3.3 Explanation of the Code

This section elaborates on the code, which can be found in appendix B.1, in order to create an understanding about the code for the reader. It elaborates on sections 2.2 and 3.2. The explanation is divided in failures (3.3.1), maintenance (3.3.2), costs (3.3.3), ordering (3.3.4), transfer (3.3.5), stock (3.3.6) and the PRNG (3.3.7).

3.3.1 Failures

The "OnComponentFailure" event is added to the PES upon initialization of the components and the execution of a degraded failure or replacement of a component. The generation of this failure is dependent on which state the component finds itself in.

When the component is in a functioning state, a critical (shock) failure time and a degraded failure time are generated according to their respective mean time to failure (MTTF). The lowest failure time is chosen and this determines the type of failure that is added to the PES. When the component jumps to a degraded state due to a degrade failure, a critical failure is added to the PES. The rate for jumping from the degraded to a failed state is the sum of the critical shock failure rate and critical degraded failure rate.

The actions upon "OnComponentFailure" depend on the criticality of the failure. In case of a critical failure, a corrective maintenance (CM) action is issued in case the item is in stock on the platform. "OnComponentReplacement" is then added to the PES. If there is no stock on the platform, it is checked whether an emergency transfer should be issues. Furthermore, on the shutdown of the system, the PM action "PMuponCriticalFailure" is issued, which is explained in section 3.3.2. If the failed component is repairable, it is send back to the base and a repair order "OnSpareRepair" is added to the PES. In case of a degraded failure, the PM action "OnPM-Replacement" is issued for the next PM period in case PM is available. Furthermore, if CBM is available a spare is reserved at the base for this PM action and the transfer order "OnPlatfor-mOrderArrivalSpareReserved" is added at the PES on the time of the next PM period.

Furthermore there is a special type of failure, the failure to start on demand. This function is called for when the system has to be started after a downtime. A random number between 0-1 is generated using the pseudo-random number generator (PRNG) and when this number is smaller than the FTSpfd the system does not start and another component replacement has to be issued. This is treated similarly to a critical failure.

3.3.2 Maintenance

Many of the functions in the PES are related to one of the maintenance policies used in this model. This section elaborates on those.

"OnComponentReplacement" represents the completion of a replacement at the platform, either of a failed or degraded component. The MTTF is decreased with the repair quality loss, in order to model imperfect repair and replacement. A new "OnComponentFailure" is generated and added to the PES if the system does not fail to start.

The DES consists of several PM functions. "OnPMReplacement" represents the PM action and is executed during a PM period. The repair time is generated using the PRNG and the "On-ComponentReplacement" for this item is added to the list. When this is larger than the PM period, the initial end of the PM period is deleted from the PES and a new "OnEndPMPeriod", with a time equal to the end of the replacement, is added. Furthermore, the earlier generated critical failure for this component is deleted from the PES. However, when there are not enough spares on the platform, the PM action is postponed until the next PM period and the "OnPMReplacement" is added to the PES again, a spare is reserved at base for this action and the "OnPlatformOrderArrivalSpareReserved" is added to the PES, at the time of next PM period. As mentioned in sections 2.2.5 and 3.3.1, upon shutdown of the system "PMuponCriticalFailure" takes place. The initial "OnStartPMPeriod" and "OnEndPMPeriod" are deleted from the PES and the new "OnEndPMPeriod" is added to the PES with the timestamp of $clock + PMinterval$. Furthermore, a search through the PES takes place in order to find degraded components on the platform that is shutdown. For those components the "OnPMReplacement" is called and if an "OnPlatformOrderArrivalSpareReserved" is in the PES for this component, it is deleted from the PES as well. The component replacement is postponed to the new PM period, if there is not enough stock. One could have chosen only to execute the "PMuponCriticalFailure" if there are enough parts in stock and, in that case, keep the PM periods as they are. The probability on a critical failure becomes therefore slightly higher, but on the other hand less downtime is caused by calling for PM periods, which is the reason for the author to implement it in this way.

"OnStartOverhaul" and "OnEndOverhaul" are added to the PES upon initializing of the model. During an overhaul the system is reset, as the components are replaced with complete new components are. Therefore, upon "OnStartOverhaul" the following events in the PES are deleted: "OnComponentFailure", "OnPMReplacement", "OnStartPMPeriod", "OnEndPMPeriod" and "OnComponentReplacement". Upon "OnEndOverhaul" the MTTFs of the components are reset to the initial values. Furthermore the failures for each component are generated again, as described in section 3.3.1. Lastly the new PM periods for all platforms and new overhaul period are initialized.

"OnSpareRepair", which means the completion of a repair at the workshop at the base, is treated similarly to a base order arrival with a quantity of 1. The only difference is the calculation of the costs.

3.3.3 Costs

The majority of the costs are calculated when they are called for. For example, when a CM action takes place, the function "CalcCMCosts" costs is called and the cost for a CM action is added to the subtotal of CMCosts. However, some of these costs are continuously monitored and those are the costs that are explained in this section.

The three time-dependent costs are holding costs, downtime costs and online CBM costs. These are calculated at every iteration of the DES and use the "TimeElapsed" function in order to determine the correct costs. The holding costs are calculated per platform and additionally for the base. "CalcDowntimeCosts" determines for each platform in which state it is. For a platform that is in a failed state the product of $DowntimePerT \times TimeElapsed$ is added to the downtime costs, while for a platform in a degraded state the product of $DowntimePerT \times TimeElapsed \times (1 - DegradedProduction)$ is added. The online CBM costs are added per iteration, but could also have been added at the end of the run when the total length of the simulation is known.

3.3.4 Ordering

The routine "CheckToOrderPlatform" is called every time a spare is used at the platform. If the platform has an outstanding order already, an order is not issued again. There are three different situations: the full demand can be met by the stock of the platform, the demand can be partly met or there is no stock at the base at all. When the whole demand can be met, the function "OnPlatformOrderArrival" is added to the PES, with a quantity that is equal to $PlatformOrderToLvl - Platform.Stock$. When a part of the demand can be met, the function "OnPlatformOrderArrival" is added to the PES, with a quantity that is equal to $Platform.-OrderToLvl - Base.Stock$. The remainder of the demand that is not met is added to the queue of BO. When there is no stock at the base, the whole demand is added to the queue. The routine "CheckToOrderBase" is called every time an order from the platform is issued. Similarly to orders from the platform, an order is not issued again when the base has an outstanding order already. When the stock level of the base is lower than the re-order level, "OnBaseOrderArrival" is added to the PES with a quantity equal to $BaseOrderToLvl - Base.Stock$.

"CheckEmergencyTransfer" is called for when there is a critical failure, the platform is out of stock and the option to have an emergency transfer is available. If there is no stock at the base, the function is not further executed. The time of the next order arrival is retrieved from the list. This time is used to estimate the costs for waiting for a normal transport. Furthermore the costs for having an emergency transfer are estimated. When this cost is lower than waiting for a normal transport, "OnPlatformEmergencyArrival" is added to the PES.

3.3.5 Transfers

"OnPlatformOrderArrival" increases the stock of the platform with the quantity of the order after which it is checked whether there are still failed components that are waiting for a spare part for replacement. If that is the case "OnComponentReplacement" is added to the PES and the stock of the platform is decreased by one again. At "OnPlatformEmergencyArrival" the "OnComponentReplacement" is added directly to the PES. The difference between "OnPlatformOrderArrival" and "OnPlatformOrderArrivalSpareReserved" is that in the latter case the stock at the base still has to be decreased. This could mean that, in the case the stock is 0 at the base, there is no spare arriving. This is because the spares that are reserved, are taken when they are needed to replace a failed component.

Upon "OnBaseOrderArrival" the queue with BO's is handled. This follows the same principle as explained at "CheckToOrderPlatform" in section 3.3.4.

3.3.6 Stock

The stock on each of the platforms is stored as an attribute of the platform itself and is manipulated by the functions as described in the previous sections. The physical stock on the base is in the model virtually represented as a 'normal' stock and a reserved stock. Virtually items are placed from the normal stock to the reserved stock when a degraded failure takes place and online CBM detects this failure, or when a PM action has to be postponed because there is insufficient stock. However, these items can be taken from the reserved stock again, for example, to meet the demand of another order that comes in. For reordering at the base the reserved stock is disregarded and only the normal stock has to be equal or lower than the re-order level. By reserving components the need for ordering spares at the base is detected sooner and should therefore result in a lower safety stock at the base.

3.3.7 Pseudo-Random Number Generator

The author uses a Pseudo-Random Number Generator (PRNG) for generating the random variables of section 3.4.3. The *Rnd* function, which is incorporated in VBA, is used to generate random numbers between zero and 1. The *Rnd* function uses a table of random numbers, and therefore, for any given initial seed, the same number sequence is generated. The author therefore uses the *Randomize* statement to initialize the random-number generator with a seed based on the system timer before calling *Rnd*. The output of *Rnd* is consequently used to generate numbers according to the uniform or exponential distribution, as can be seen in the code in B.1.

3.4 Quantities of the Model

This section describes the quantities that are used in this model. Their respective parameters are given and their values for the Kristin case are given as well. The quantities are divided in decision variables (3.4.1), output variables (3.4.2), random variables (3.4.3) and constants (3.4.4).

3.4.1 Decision Variables

The model tries to optimize several decision variables. These decision variables are, however, different in the two different simulations we run. The decision variables for the Kristin case are the (s, S) spare order policy variables, for both the platform and base. For the joint optimization the decision variables are extended with the PM interval, overhaul interval and the boolean CBM for whether online CBM takes place or not. These variables are constants, when optimizing the Kristin case, for which the values are shown in table 3.3. The decision variables for the Kristin case can be found in table 3.2.

It has to be stated that the initial stock levels could also be chosen as decision variables. However, the author assumes that these are equal to the order-to-levels, as this is the most logi-

Table 3.2: Decision Variables for the Kristin Case

Decision Variable	Parameter
Base S	BaseOrderToLvl
Base s	BaseOrderLvl
Platform S	PlatformOrderToLvl
Platform s	PlatformOrderLvl

Table 3.3: Additional Decision Variables for the Joint Optimization

Decision Variable	Parameter	Value
PM interval	PMInterval	2000
Overhaul interval	OverhaulInt	10000
Online CBM	CBM	TRUE

cal initial state. Therefore they are not real decision variables. The same holds for the threshold value for PM at the platform. This one could be regarded as a decision variable and optimized, but the author choses to have a fixed value for this threshold.

3.4.2 Output Variables

The model has two output variables: the total amount of costs and the average availability. The total amount of costs is used as the objective function for optimizing the model, where this variable is to be minimized. The purpose of the average availability is merely to show the performance of the system. One could opt for having the average availability, or a combination of total costs and average availability as the objective function. The author choses not to do this, as the simulations show that, with the input parameters of the Kristin case, the availability is not varying much for different combinations of the decision variables. Table 3.4 summarizes the output variables.

Table 3.4: Output Variables of the Model

Output Variable	Parameter
Total costs	AverageTotCosts
Average availability	AvgAvailability

3.4.3 Random Variables

The model consists of several random variables, which are summarized in table 3.5. These random variables cause the randomness in the model and the values are generated in the simulation using a PRNG, following the distribution of the random variable.

The Kristin platform is being supplied with spare parts, personnel and food supplies by a boat twice a week. Approximated this means that the maximum time for a transfer to arrive after the order is placed, is 84 hours. The author assumes that it must take some time to prepare a transport. Therefore the author argues that this transfer times follows a uniform distribution, with as minimum the order handling time, and as maximum the sum of the order handling time and the time between two supplies.

The failure rates of the components that are modeled, are likely to follow a Weibull distribution with a value for α larger than one. As described in section 2.2.3, the failures are modeled as a Markov model, which requires exponential transition rates between the states. Furthermore, SINTEF (2009) presents the failure rate data in an exponential distribution. Using the model of Hokstad and Frøvig (1996), as introduced in 2.2.3, and the information of SINTEF (2009), we come to a degraded-shock ratio of 0.73 – 0.28. The overall critical failure rate is then assigned to critical shock and critical degraded failure rate according to this ratio. The failure rates are divided by a factor of 5, since the failure rates in SINTEF (2009) are for the complete gas turbine, while here it is assumed it consists of 5 identical maintainable items. We then obtain the failure rates as given in table 3.5.

The replacement and repair rates are assumed to be exponentially distributed as well. The replacement rates represent the rate for replacing a component at the platform, while the repair rates represent the rate for repairing a repairable component at the workshop at the base. The replacement rate is based on the very limited data from Statoil, as this is the author's best approximation. The repair rates are based on SINTEF (2009).

3.4.4 Constants

Besides all these variables, the model consists of various constants. The values of these constants, however, can be changed for each simulation according to the wishes of the user. Con-

Table 3.5: Random Variables of the Model

Random Variable	Parameter	Distribution	Value
Transfer Time Base-Platform	OrderHandlingTime,transTime	Uniformly	4-88
Degraded failure rate	λ_d	Exponentially	0.000052
Critical degraded failure rate	λ_{cd}	Exponentially	0.000034
Critical shock failure rate	λ_{cs}	Exponentially	0.000013
Replacement rate degraded	μ_d	Exponentially	0.25
Replacement rate critical	μ_c	Exponentially	0.08
Repair rate degraded	γ_d	Exponentially	0.0556
Repair rate critical	γ_c	Exponentially	0.0385

stants like these are, for example, the number of runs per simulation or the various costs. Table 3.6 summarizes these constants with according parameters and used values.

The design life of the Kristin platform is 25 years, which is approximated by 219000 hours. The number of runs is set on 400, which results in a computation time of approximately one minute per simulation. The desired margin of error (E) of formula 2.1 is set on 0.5% of the average and a confidence interval with $\alpha = 0.05$ is chosen, which results in approximately 391 runs. This means that we are 95% sure that the results are accurate within ±0.5%.

The component in this case is a non-repairable component and the fail to start on demand probability is taken from SINTEF (2009).

Preventive maintenance (PM) takes place every 2000 hours. It is assumed that the minimum duration is 4 hours, for testing and controlling of the equipment. The actual PM interval can be higher than these 4 hours, when preventive replacements have to be conducted that take more than these 4 hours. Every 10000 hours overhaul takes place, for which the duration is 6 hours.

The repair quality, which can be seen as the replacement quality in this case, since new items are ordered instead of repaired, is assumed to be 0.99. This means that the $MTTF$ decreases with 1% upon each component replacement, this due to possible non-optimal installation of the component.

It is assumed that at least one spare should be in stock for using that component for replacement of a degraded component. Hence, this is not an extra restriction on PM, since it is not known what the policy of Statoil is in this case.

The author assumes the majority of the costs, as there is no data available for these costs. Logical assumptions are made, such as holding costs for the platform are higher than for the base and emergency transfer costs are much higher than regular transfer costs. These regular transportation costs are assumed to be low, since these deliveries take place regardless of the need for transporting the spare. The downtime costs for the Kristin platform are approximately 15 million kroner a day. The costs per item are based on spare part information from Statoil. An average from the main maintainable items of the gas turbine is taken for the cost price of the item.

Three different situations are simulated in order to determine the impacts of having different PM strategies on stock levels:

1. Overhauls that take place according to BRP.
2. Overhauls that take place according to BRP, and offline CBM that takes place with intervals according to ARP.
3. Overhauls that take place according to BRP, offline CBM that takes place with intervals according to ARP and online CBM.

The constants PM, Overhaul and CBM are therefore varying during the three different simulations.

Table 3.6: Constants of the Model

Constant	Parameter	Value
#Platforms	nPlatforms	5
#Components per Platform	nComponents	5
Design life platform (h)	MaxTime	219000
Number of runs	nRuns	400
Emergency transfer option	emergTrans	TRUE
Emergency transfer time (h)	emergTransTime	12
Order from Base time (h)	OrderTime	720
Degraded production (0-1)	DegradedProduction	0.95
Fail to start on demand	FTSpfd	0.0034
Repairable	Repairable	FALSE
Platform initial stock	PlatformStockLvl	PlatformOrderToLvl
Base initial stock	BaseStockLvl	BaseOrderToLvl
PM	PM	
PM duration (h)	PMDuration	4
Overhaul	Overhaul	
Overhaul duration (h)	OverhaulDur	6
Repair Quality (0-1)	RepQual	0.99
Min. stockLvl at Platform for PM	PMminstock	1
Transfer Base-Platform per item	CostPerTransfer	400
Emergency transfer per item	CostPerEmergencyTrans	750000
Repair of item at base	CostPerRepair	1000
PM replacement per item	CostPerPMReplacement	2000
Start PM period	CostPerPMPeriod	2500
Downtime per hour	DowntimePerT	625000
CM replacement per item	CostPerCM	6000
Holding per item per hour on platform	HoldingPlatformPerT	2
Holding per item per hour on base	HoldingBasePerT	1
Order cost per item	OrderCostPerItem	10000
Cost per order on base	CostPerBaseOrder	500
Cost per order on platform	CostPerPlatformOrder	250
Cost per overhaul per component	CostPerOverhaul	10000
Cost for online CBM per time unit	CBMpert	2

Chapter 4

Optimization Methods

This chapter introduces methods that can be used for optimizing a DES. Sections 4.1, 4.2 and 4.2.2 focus on optimizing the decision variables of the DES, while 4.3 focuses on optimizing the computation time of the DES.

4.1 Introduction to Optimization Approaches

This section gives a brief overview of which some methods one can use to find the optimal values of the model, in order to find optimal stock values in the Kristin case and a method for joint optimization of the PM interval, the overhaul interval, spare order threshold and stock levels.

In a relatively small case, with a maximum S of 10 for both the base we have $10! \times 10! = 1.3 \times 10^{13}$ search spaces. Finding the optimal value by calculating all these options would take too much time and therefore there is a need for a faster method. Firstly, the author develops a trial and error method which should decrease the computation time significantly. All possible combinations of base and platform stock levels (s, S) with $S = 1, 2, ..., 10$ and s values in the range from $s = max(0, S-2)$ to $s = S-1$ are simulated for each of the simulations. After these simulations, the behavior of the cost function becomes clear and a local optimization strategy follows consequently through which the local minimum values can then be found. When it is clear that an S-value of 10 is not sufficient and it is increased. The values of the parameters s and S are either increased or decreased by 1 for each simulation. If an increment of one of the parameters yields a higher cost, we know that the previous value was the local maximum and further increments do not lead to better results. The same holds for decreasing of one of the parameters. Through this approach al the local minimum values that could be the global minimum cost value are found and therefore this methods yields a value which is close to the global minimum cost value. This method is more or less a trial and error method, which is undesirable. It furthermore does not only take a large computation time, but it also requires a lot of time from the designer to enter the input. The designer should not be a mediator between the model and the algorithm that optimizes the model. One should desire a simulation that automatically finds the optimal values. Therefore the author proposes the use of another method for optimizing the model.

The earlier performed literature study by the author describes both marginal analysis (MA) as genetic algorithms (GA) as possible ways of optimizing a DES. The author sees a MA here as unfit, since it is unclear which stock values follow each other. There are no clear increments with value of one in the parameters. An MA is useful when we want to optimize an $(S-1, S)$ policy, since the S has increments of one and the optimal value can easily be found. With an (s, S) policy this is not possible, since there is not such a similar linear increase of the parameters (s, S).

The author therefore choses to use a GA for optimizing this model. An introduction to GA can be found in Yu and Gen (2010), as it is not further explained here. The main advantage of using a GA for a complex DES model is that it works very efficient in a situation where there exists a lot of local minima/maxima. Since the trial and error method shows a lot of local minima, developing a GA seems like an efficient way in optimizing this problem. Paul and Chanev (1997) use a simplified GA for optimizing a complex DES model. The algorithm that the author codes is based on their algorithm. This modified algorithm, which is based on a highly disruptive crossover and elitist selection, has proved to be a good alternative to the classical GA (Paul and Chanev, 1997).

4.2 Genetic Algorithm for Optimizing Decision Variables

4.2.1 Introduction to Genetic Algorithm

The first step is the **Initialization**. The author decides to create a population size *popL* of 100. This size should be big enough to ensure the variance in the population after several iterations, but is not so big that initialization takes too much computation time. Furthermore, with a very large population size it requires more iterations to develop the initial population towards a population with a higher fitness level. During this initialization the order-to-levels S for both the base and platform are initially computed by using the PRNG that generates numbers according to a uniform distribution, with a minimum of 1 and maximum S value of 15 for both base and platform. Based on the trial-and-error method this should be sufficient to find the optimal stock values. After initialization of the order-to-levels, the values for the re-order levels are generated. These are generated using the same PRNG, but with a minimum of 0 and a maximum of the according order-to-level S minus 1. The (s,S) order policy of the base represents one gene and the order (s,S) order policy of the platform represent another gene. Consequently the DES is executed for all the individuals in the population in order to find the total costs of the respective stock levels.

After the initialization, **Evaluation** takes place. The goal function in this optimization is to *minimize the total costs*. After this evaluation the weakest individual, which is the one that yields the highest costs, dies and is replaced by a new individual.

This new individual is created by **Crossover**. Two parents are randomly chosen for this crossover. The order policy (s,S) for the base and platform are chosen with a probability *Ppar* from the fittest parent and a probability of *1-Ppar* from the second parent. After the crossover mutation can take place in order to get randomness in the population.

This **Mutation** can take place on this new offspring with a probability of *Pmut*. When mutation takes place, one of the two genes is randomly chosen and changed. The chosen gene is regenerated according to the same principle as the initialization.

The last step is **Replication** of the best individual, which takes place with a probability of *Pbest*. The best individual is replicated and added to to the population, replacing the weakest individual again. This process is being repeated for a number of predefined iterations. A summary of the input parameters is given in table 4.1.

Table 4.1: Input Variables for the GA for Optimization of the Model

Parameter	Variable	Value
popL	Population size	100
Ppar	Probability that the gene of the fitter parent is chosen during crossover	0.7
Pmut	Probability of mutation of a gene after crossover	0.5
Pbest	Probability of replication of the best individual at each iteration	0.4
nIterations	Number of iterations before termination of the algorithm	50

4.2.2 Genetic Algorithm for Joint Optimization

The GA can be further extended in order to optimize jointly the PM interval, the overhaul interval, spare order threshold and stock levels. The principles of the algorithm stay the same, but during this optimization problem there are not 4 variables, but 7 decision variables we should optimize. These variables and their possible ranges are summarized in table 4.2. There are more variables to optimize and therefore the search grid increases significantly. Therefore other input variables for the model are necessary, which can be found in table 4.3. The number of iterations is increased, since there is a bigger search grid to cover. Therefore the value of $Pbest$ is decreased, otherwise the population would loses it variety as too many copies of the best individual would be made.

Table 4.2: Decision Variables for Joint Optimization of the Model

Parameter	Variable	Range
S-base	Order-to-level at the base	s-base+1 - 15
S-platform	Order-to-level at the platform	s-platform+1 - 15
s-base	Re-order level base	0 - S-base-1
s-platform	Re-order level platform	0 - S-platform-1
Overhaul-interval	Time in-between overhauls	PM interval - 219000
PM-interval	Time in-between tests	1000 - Overhaul-interval
Ls	Spare order threshold level	Degraded - Failed

It is assumed that offline condition monitoring occurs more often than overhauls of the system and therefore this interval cannot be higher than the overhaul interval. When the PM and overhaul intervals become too small, the computation time would increase significantly, and therefore a minimum value of 1000 hours is assumed. The design life is the maximum of the PM

Table 4.3: Input Variables for the GA for Joint Optimization

Parameter	Variable	Value
popL	Population size	100
Ppar	Probability that the gene of the fitter parent is chosen during crossover	0.7
Pmut	Probability of mutation of a gene after crossover	0.5
Pbest	Probability of replication of the best individual at each iteration	0.3
nIterations	Number of iterations before termination of the algorithm	100

and overhaul intervals, which means that no preventive maintenance actions take place. Both intervals are changed by factors of 100 hours in order to decrease the search space volume and to make changes in these intervals significant.

Since components only have three states (functioning, degraded, failed), the threshold level can only either be "degraded" or "failed". However, it is only natural to order a spare part when a preventive maintenance action takes place, this spare order threshold level is approached by having online CBM or not. That means that if there is online CBM, a spare order is send out upon a degraded failure. When there is no online CBM, the degraded failure is only detected during a testing period. Hence the spare part is not ordered when the degraded failure occurs.

4.3 PES Handling Optimization

As the simulation requires a significant amount of computation time, one should not only try to optimize the output results, but also the computation time. Code should be written such that the events are executed as efficient as possible. One way of optimizing computation time is the method for the handling of the pending event set (PES). This section explains four different methods, which are tested according to four different cases in order to determine their functionality.

The first two methods are based on Vatn (2012), who uses a list that is stored in an array. The first method uses a linear search through this array, while the second method uses an indexed list, which enables fast access to the PES. In the second method the program first searches through the indexed list, which points to some items in the PES. Through the search in the index list one does not have to search the complete PES. Listing 4.1 shows the core of method 1, while listing 4.2 shows the core of method 2. Using a fixed length for the array requires quite some memory space, but it has as main advantage that the *ReDim* statement does not has to be used every time an event is being added to or deleted from the list. This *ReDim* statement requires a lot of computation time, as it copies the whole array to another memory space every time it is being called (Getz and Gilbert, 2000). The method with a variable array length with resizing of the array using the *ReDim* function is not being tested here.

Listing 4.1: Method 1

```
Type PES1Element
t As Single
NextElement As Integer
NextAvail As Integer
End Type
Const MaxDim As Integer = 4096
Public PES1(1 To MaxDim) As PES1Element
```

Listing 4.2: Method 2

```
Type PESElement
t As Single
NextElement As Integer
NextAvail As Integer
pIndx As Integer
End Type
Const MaxDim As Integer = 4096
Public PES(1 To MaxDim) As PESElement
Type IndxElement
t As Single
pPES As Integer
End Type
Const SizeOfIndx As Integer = 200
Public Indx(1 To SizeOfIndx) As IndxElement
```

Getz and Gilbert (2000) explain another method that uses a linked list class. It is this class that the author has used for implementation of the tool, as the code is clear and easy to understand. This method has as advantage over the previous two methods, that the use of the memory varies according to the length of the list. There is therefore no need to reserve a piece of the memory for the array. One can use different search algorithms. The author uses a linear search method for the implementation of this module, but also a binary search tree is tested. These linked classes are initialized by setting a "listhead" or "treehead" to the first event that is to be inserted. The next events that are inserted are then linked through the "NextItem" or "LeftChild" and "RightChild" attributes. Listing 4.3 shows the core of method 3, while listing 4.4 shows the core of method 4. The search algorithms are not shown here.

Listing 4.3: Method 3

```
' ListItem class.

Public t As Single
Public NextItem As ListItem
```

Listing 4.4: Method 4

```
' TreeItem Class.

Public t As Single
Public LeftChild As TreeItem
Public RightChild As TreeItem
```

These four different methods are tested for four different cases, which are shown in table 4.4. A PES length of 35 is chosen, since this the average length of the PES in the Kristin case. Additionally, a PES length of 100 is chosen in order to determine the quality of the methods in different lengths of the PES. Each length is tested with 500 and 2000 iterations in order to determine the relation between the initialization speed of the PES and the search and delete speed of the PES. 10 Sets of times are generated and are used for testing these cases. The author choses to generate 10 different sets and use the same sets for each of the cases in order the reduce the influence of the generated times on the computation time of each case. For each set the average of 100 runs is taken as actual computation time for that specific combination of time set and case.

Table 4.4: Simulation Cases for PES Handling Optimization

Case	Length	Iterations
1	35	500
2	100	500
3	35	2000
4	100	2000

Chapter 5

Results

This chapter discusses the results of the optimization problems in this thesis. Section 5.1 discusses the results for the Kristin Case, while the results for the use of the GA for the joint optimization of the variables are discussed in section 5.2. Lastly, section 5.3 discusses the results for the efficiency of the different PES methods.

5.1 Optimal Stock Values for the Kristin Case

The model is used for simulating the three cases that are explained in section 3.4.4., in order to determine the impact of different PM strategies on the stock levels. Table 5.1 shows the results for the optimization of the case by trial-and-error method, while the results for the optimization by the GA are given in table 5.2. The computation time for the trial-and-error method is approximately four days, while the computation time of the model by the GA is approximately one day.

The lowest total costs are for the case where we have online CBM, while the highest costs are for the case where we only have overhauls. Case 2, with offline condition monitoring and overhauls gives results that are really close to case 1. This is something in line with the expectations. The amount of spare parts in stock for case 3 are the lowest, while they are the highest for case 1. This can be explained that more PM actions take place, and therefore more spares are necessary. As mentioned in the introduction, these could in theory be supplied with a JIT policy. This is not always possible, since PM actions also take place when the system shuts down after a critical system, which cannot be planned.

Table 5.1: Results for Optimization of the Model by Trial-and-Error

Variables	Case 1	Case 2	Case 3
platform (s,S)	(6,9)	(4,5)	(3,4)
base (s,S)	(4,6)	(2,5)	(3,8)
Tot. costs (10^9 kr)	9.442	9.446	18.27
Availability	0.996	0.996	0.998

Table 5.2: Results for Optimization of the Model by the Genetic Algorithm with 400 runs

Variables	Case 1	Case 2	Case 3
platform (s,S)	(7,9)	(7,8)	(0,1)
base (s,S)	(10,13)	(8,11)	(3,4)
Tot. costs (10^9 kr)	9.433	9.427	18.25

The results for the optimization by the GA give a platform stock policy of only $(0,1)$ for the 3rd case, which seems to be too low, even though the total costs are lower than what we had obtained by the trial-and-error method. For case 1 and case 2 the stock levels are much higher with optimization by GA than with optimization by trial-and-error. These differences can be explained by the fact that 400 runs give results that are not accurate enough. The optimal policies of the GA optimization are simulated once more and then much higher results, which were far from optimal, are obtained for these policies. These outliers are caused by the low amount of runs, which have high impact while optimizing the model by the GA. Since the search grid is significantly smaller than during the trial-and-error-method, we can easily afford it to use more iterations to obtain more accurate results. The margin of error is therefore set on 0.1%, which leads to 10000 runs and a computation time of approximately 25 minutes per fitness calculation of the individual. The results for the simulation with 10000 runs are shown in table 5.3. Despite having 25 times the amount of runs than the simulation with the trial-and-error method, the computation time is still short with a length of approximately 2.5 days.

Table 5.3: Results for Optimization of the Model by the Genetic Algorithm with 10000 runs

Variables	Case 1	Case 2	Case 3
platform (s,S)	(0,3)	(12,13)	(2,3)
base (s,S)	(2,5)	(4,12)	(8,9)
Tot. costs (10^9 kr)	9.471	9.489	18.36

The total costs for each of the cases are slightly higher for the results of the GA than for the trial-and-error method. This is caused by the fact that the GA does not guarantees to find the optimum value, but a local optimum that is close to the global optimum value. Especially case 2 gives completely different values for the re-order and order to levels. This can be caused that this local optimum is very close to the global optimum value, even though the values of the variables are very different. This phenomenon is already discovered by the author while executing the trial-and-error method. Also by this optimization, the difference between costs for case 1 and case 2 is very small. Adding online CBM seems to have a positive effect on the costs, however, this is a very small effect. This can be explained by the fact that more PM actions are executed, as the probability of having a spare part on the platform when this action has to take place is higher as a result of the reservation system. Therefore more spares are used throughout the lifetime of the platform, which increases the total costs. The decrease in costs related to a failure is only slightly smaller than the increase in costs.

As the results for the decision variables are differing, it is hard to get conclusions about the impact of the different PM policies. The author believes that this is not exactly clear, because PM periods also take place upon a critical failure and follow an ARP policy, and not a BRP. Therefore it is hard to predict the spares that are necessary for PM actions. The demand for spares for PM is uncertain and therefore the amount one needs in stock increases, which is contrary to what one might expect. It is normally expected that having PM actions can reduce the safety stock. However, these results show that it is not necessarily always the case.

5.2 Joint Optimization of the Decision Variables

The Genetic Algorithm of section 4.2 is used for joint optimization of the decision variables, for which the results can be found in table 5.4. The computation time is approximately 3.5 days.

The costs are significantly reduced in comparison with the results from section 5.1. The PM interval is significantly decreased, while the overhaul interval has been increased. As the results from the previous section suggest, there is not a big difference between having online CBM or not. The results in this optimization say there should not be CBM. The stock levels are rather low, especially for the platform, but this can be explained by the low PM interval. The probability of having to replace multiple components in a PM period and the probability of having a critical failure become small and therefore less stock is necessary.

Table 5.4: Results for Joint Optimization of the Model by the Genetic Algorithm

Variables	Value
platform (s,S)	(0,2)
base (s,S)	(2,4)
PM interval (h)	1100
Overhaul interval (h)	12400
CBM	FALSE
Tot. costs (10^9 kr)	8.139

5.3 Efficiency of PES Methods

Table 5.5 shows the average of the computation time of the 10 different time sets. However, to be able to compare the performance, the relative times are computed, which are shown in table 5.7.

Table 5.5: Results for Efficiency of PES Methods: Actual Times (in seconds)

	Method 1	Method 2	Method 3	Method 4
Case 1	0.052645	0.053598	0.057199	0.059488
Case 2	0.055965	0.056371	0.062170	0.063246
Case 3	0.203449	0.205215	0.218781	0.228121
Case 4	0.206082	0.207730	0.229848	0.233000

Table 5.6: Results for Efficiency of PES Methods: Relative Times (in %)

	1vs2	1vs3	1vs4	2vs3	2vs4	3vs4
Case 1	1.810	8.652	13.000	6.720	10.990	4.002
Case 2	0.726	11.887	13.010	11.080	12.196	1.004
Case 3	0.868	7.536	12.127	6.611	11.162	4.269
Case 4	0.800	11.532	13.062	10.647	12.165	1.371

It is clear that method 1 is the fastest one, regardless of the case. However, when analyzing the relative times, we learn that the more complex structures (method 2 and 3), perform relatively better when the length of the PES is increasing. Table 5.7 furthermore shows that when the amount of iterations increases, the method the author uses (method 3), is performing relatively better than methods 1 and 2. This means that initializing the linked list of method 3 is slower than initializing the array of methods 1 and 2, while the deleting and inserting of items might go faster. Therefore, the author choses to run another test with 20000 iterations for both PES lengths. Table 5.7 shows the results for this simulation, relative to method 1.

Table 5.7: Results for efficiency of PES methods with 20000 iterations: relative times (in %)

	Method 2	Method 3	Method 4
Length: 35	1.212	6.180	6.972
Length: 100	-0.308	7.653	7.810

The results show that method 1 is not the fastest, when the PES length is 100 and the simulation runs with 20000 iterations. Furthermore, the performance of both methods 3 and 4 becomes better when the number of iterations increases in comparison with method 1. The author believes that method 4 performs slower than any other method, because it loses time for deleting the first item from the PES. Deleting the first item from the tree requires some steps to find it, as it is placed in a branch of the three, while for the other three methods the item that has to be deleted, is the first item in the list. These tests show that the benefit of the faster search for inserting items in the tree are not outweighing the loss of speed when deleting an item.

Chapter 6

Summary and Recommendations for Further Work

This final chapter summarizes and concludes the work of this thesis in section 6.1, while these are briefly discussed in section 6.2. Recommendations for further work are given in section 6.3.

6.1 Summary and Conclusions

Section 2.1 shows several possibilities for determining the optimal stock values. The author shows that developing a DES in VBA is a good option for solving this problem. The author determines the optimal stock values for the Kristin case by a trial-and-error method, for which the results are shown table 5.1. Table 5.3 shows the results for the Kristin case by solving it with the genetic algorithm the author has created in section 4.2. These results furthermore show the impacts of the different PM strategies on the total costs and stock levels. The total costs decrease significantly when PM is scheduled. The difference in total costs between having online CBM or not is rather small. Unfortunately, as there are many local optima that are very close to the global optimum, it was very difficult to find a consistent answer for the impact of the different PM strategies on stock levels. The author argues that enabling PM upon critical failures and following an ARP strategy might increase the amount of spares in stock, rather than decrease, but this has not been tested extensively.

Section 5.3 shows the efficiency results for the different methods for handling a PES, which are introduced in section 4.3. This research shows that easy constructions, like a linked list, are most efficient when a PES has a short length and a low amount of iterations take place. When the length and number of iterations increases, a more complex constructions, such as a linked list combined with an index list, can become faster than the easy constructions. It can therefore be concluded that the designer of the DES should determine the characteristics of the DES, before an appropriate PES method can be chosen.

The author shows that a simplified genetic algorithm, which is introduced in section 4.2, can be used for solving the model. The decision variables can be optimized and the genetic algorithm can jointly optimize the PM interval, the test interval, spare order threshold and stock levels. Especially for the joint optimization the genetic algorithm gives good results, which are shown in table 5.4. The total costs are reduced significantly using this joint optimization in respect to the initial values of that are used in the Kristin case. The simplified genetic algorithm has as main advantage over a normal genetic algorithm that it is much easier to understand and implement. As good results are obtained in this thesis, it is shown that the proposed simplified genetic algorithm is a good alternative for a normal genetic algorithm.

6.2 Discussion

As stated in section 1.3, the data acquisition for the Kristin case is very limited and results for this case are therefore not really useful for practical purposes. The output should therefore not be used in real life.

The testing of the speed of the methods only tests the speed of inserting items in the PES and retrieving the next item from the PES. However, as the explanation of the code in 3.3 shows, there are also functions that are searching for specific items in the list and consequently delete these elements. These are not included in this research, as only the two basic functions are tested here. Therefore, the overall performance for each of the methods in might be slightly different with respect to this specific model.

Due to the characteristics of the model, optimizing the spare order threshold with online CBM, is too simplistic in this model. As there is only one state between a functioning and failed state, the degraded state, there is no freedom in choosing the spare order threshold.

The results of the simulation of the simplified genetic algorithm might be better if the input variables of the algorithm are optimized. The author only runs some trial runs in order to find good values for these. With other values for the variables the algorithm might find a local optimum that is closer to the global optimum or the computation time might be reduced.

6.3 Recommendations for Further Work

The impacts of having different PM strategies on stock levels is recommended for further work. One could test the differences in spares when having PM upon critical failures or not, and when the ARP or BRP policy is chosen.

The research to the efficiency of handling the PES should be extended to all possible manipulations of the PES, not only the two main functions. The author therefore recommends further research that includes more manipulations, like, for example, deleting specific items from the list. Furthermore, these methods could be compared with other methods for keeping a PES. Furthermore, the amount of memory the methods take is not tested. This is also a valuable characteristic, so this should be included in further research.

A model where online CBM is used more extensively should be developed. A model with more states could be created, or a model that where components' performance are represented with a continuously variable, rather than with discrete states.

The simplified genetic algorithm that is used in this thesis needs some further research in order to give even better results. One can develop a method in order to optimize the input values the algorithm requires.

Appendix A

Acronyms

ATP	Age-Replacement Policy
BO	Backorders
BRP	Block-Replacement Policy
CBM	Condition-Based Maintenance
CM	Corrective Maintenance
CPN	Colored Petri Nets
DES	Discrete-Event Simulation
FIFO	First-In-First-Out
GA	Genetic Algorithm
JIT	Just-In-Time
MA	Marginal Analysis
MI	Maintenance Items
MTTF	Mean Time To Failure
NTNU	Norwegian University of Science and Technology
PES	Pending-Event Set
PM	Preventive Maintenance
PRNG	Pseudo-Random Number Generator
RAMS	Reliability, Availability, Maintainability and Safety

Appendix B

Code

This appendix shows the VBA code for the DES Tool and for the Genetic Algorithm.

B.1 DES Tool

This section contains the codes of the different modules of the DES Tool. The last two listings contain the code for the creation of the list and queue class. The list class is used as the PES of the simulation, while the queue is used to store backorders with a FIFO policy.

Listing B.1: Main Module

```
Public totCost As Single
Public Avail As Single
Public transTime As Single
Public emergTransTime As Single
Public emergTrans As Boolean
Public listHead As ListItem
Public Clock As Single
Public qFront As QueueItem
Public qRear As QueueItem
Public AverageTotCosts As Single
Public AvgAvailability As Single
Public OrderTime As Single
Public DegradedProduction As Single
Public OrderHandlingTime As Single
```

```
Dim MaxTime As Single
Dim Data As Variant

Sub MainProgSimul()
'Main program that runs the simulation nRuns times
Dim nRuns As Integer
AverageTotCosts = 0
AvgAvailability = 0
nRuns = Worksheets("SpareSimulation").Range("nRuns").Value
MaxTime = Worksheets("SpareSimulation").Range("LifetimePlatform").Value

For i = 1 To nRuns
    SubProgSimul
    CalcTotCosts
    GetAvailability
Next

'Results are written in the sheet
Worksheets("SpareSimulation").Range("TotalCosts").Value = (AverageTotCosts / nRuns)
Worksheets("SpareSimulation").Range("Availability").Value = (AvgAvailability / nRuns)
End Sub

Private Sub SubProgSimul()
'Core of the simulation
Dim toclearlist As Boolean

InitVariables
InitPlatforms
InitComponents
InitCosts
If PM Then
InitPM
End If
If Overhaul Then
InitOverhaul
End If
```

```

Do While MaxTime > GetClock()
    Data = GetNxtEvent()
    CalcHoldingCosts
    CalcDowntimeCosts
    CalcUpTime
    CalcCBMcosts
    ExecuteCallback Data
Loop

'Clear the list and queue to create memory space.
toclearlist = True
Do While toclearlist
    ClearList toclearlist
Loop
Do Until IsEmpty()
    ClearQueue
Loop
End Sub

Function InitVariables()
Set listHead = Nothing
Set listCurrent = Nothing
Set listPrevious = Nothing
Set qFront = Nothing
Set qRear = Nothing
Clock = 0
PrevClock = 0
CompNumb = 0
PlatNumb = 0
FC = 0
MTTF = 0
OrderHandlingTime = Worksheets("SpareSimulation").Range("OrderHandlingTime").Value
OrderTime = Worksheets("SpareSimulation").Range("OrderTime").Value
transTime = Worksheets("SpareSimulation").Range("TransferTime").Value
emergTransTime = Worksheets("SpareSimulation").Range("EmergencyTransferTime").Value
```

```
emergTrans = Worksheets("SpareSimulation").Range("EmergencyTransfer").Value
MDTd = 1 / Worksheets("SpareSimulation").Range("MDTd").Value
MDTc = 1 / Worksheets("SpareSimulation").Range("MDTc").Value
MTTRd = 1 / Worksheets("SpareSimulation").Range("MTTRd").Value
MTTRc = 1 / Worksheets("SpareSimulation").Range("MTTRc").Value
FTSpfd = Worksheets("SpareSimulation").Range("FTSpfd").Value
PMminstock = Worksheets("SpareSimulation").Range("PMminstock").Value
PM = Worksheets("SpareSimulation").Range("PM").Value
Overhaul = Worksheets("SpareSimulation").Range("OVERHAUL").Value
RepQual = Worksheets("SpareSimulation").Range("RepQual").Value
PMInterval = Worksheets("SpareSimulation").Range("PMInterval").Value
PMDuration = Worksheets("SpareSimulation").Range("PMDuration").Value
CBM = Worksheets("SpareSimulation").Range("CBM").Value
DegradedProduction = Worksheets("SpareSimulation").Range("DegradedProduction").Value
End Function

Function ClearList(toclearlist As Boolean)
'Clear the list
If listHead Is Nothing Then
    toclearlist = False
    Exit Function
End If

Set listCurrent = listHead.NextItem
If listCurrent Is Nothing Then
    Set listHead = Nothing
Else
    Set listHead = listCurrent.NextItem
End If
End Function

Function ClearQueue()
'Clear BO queue
If qFront Is qRear Then
    Set qFront = Nothing
    Set qRear = Nothing
```

```
Else
    Set qFront = qFront.NextItem
End If
End Function
```

Listing B.2: PES Module

```
Dim PrevClock As Single
Public UpTimes() As Single
Public Availabilities() As Single

Function GetNxtEvent() As Variant
Dim t As Single

Set listCurrent = listHead

t = listCurrent.t
PrevClock = Clock
Clock = listCurrent.t

GetNxtEvent = listCurrent.Data
DeleteElementFromList t
End Function

Function ExecuteCallback(Data As Variant)
Dim FuncName As String
Dim CompOrPlat As Integer
Dim Quantity As Integer
QorFC = Data(2)
CompOrPlat = Data(1)
FuncName = Data(0)

Select Case FuncName
    Case "OnComponentFailure"
        OnComponentFailure CompOrPlat, QorFC
    Case "OnComponentReplacement"
        OnComponentReplacement CompOrPlat
```

```
    Case "OnSpareRepair"
        OnSpareRepair
    Case "OnPlatformOrderArrival"
        OnPlatformOrderArrival CompOrPlat, QorFC
    Case "OnBaseOrderArrival"
        OnBaseOrderArrival QorFC
    Case "OnPMReplacement"
        OnPMReplacement CompOrPlat
    Case "OnStartPMPeriod"
        OnStartPMPeriod CompOrPlat
    Case "OnEndPMPeriod"
        OnEndPMPeriod CompOrPlat
    Case "OnStartOverhaul"
        OnStartOverhaul
    Case "OnEndOverhaul"
        OnEndOverhaul
    Case "OnPlatformEmergencyArrival"
        OnPlatformEmergencyArrival CompOrPlat
    Case "OnPlatformOrderArrivalSR"
        OnPlatformOrderArrivalSR CompOrPlat, QorFC
End Select
End Function

Function CallBackData(CallBackFunction As String, Optional ByVal Element As Integer = 0,
    Optional ByVal Element2 As Integer = 0)
' This function combines the function name with a parameter for easy retrieving
CallBackData = Array(CallBackFunction, Element, Element2)
End Function

Function InsertElementInList(t As Single, Data As Variant)
Dim listNew As ListItem

Set listNew = New ListItem
listNew.t = t
listNew.Data = Data

```

```
SearchList t, listCurrent, listPrevious

If Not listPrevious Is Nothing Then
    Set listNew.NextItem = listPrevious.NextItem
    Set listPrevious.NextItem = listNew
Else
    Set listNew.NextItem = listHead
    Set listHead = listNew
End If
End Function

Function DeleteElementFromList(t As Single)

SearchList t, listCurrent, listPrevious

If listPrevious Is Nothing Then
    Set listHead = listCurrent.NextItem
Else
    Set listPrevious.NextItem = listCurrent.NextItem
End If
End Function

Function SearchList(ByVal t As Single, ByRef listCurrent As ListItem, ByRef listPrevious
    As ListItem)
Set listPrevious = Nothing
Set listCurrent = listHead

Do Until listCurrent Is Nothing
    If t > listCurrent.t Then
        Set listPrevious = listCurrent
        Set listCurrent = listCurrent.NextItem
    Else
        Exit Do
    End If
Loop
End Function
```

```
Function EventNotice(t As Single, Data As Variant)
EventNotice = InsertElementInList(t, Data)
End Function

Function GetClock()
GetClock = Clock
End Function

Function TimeElapsed()
TimeElapsed = Clock - PrevClock
End Function

Function CalcUpTime()
'Keeps track of the uptime of each platform
For i = 1 To nPlatforms
    If Not Platforms(i).State = Failed Then
    UpTimes(i) = UpTimes(i) + TimeElapsed
    End If
Next
End Function

Function GetAvailability()
'Calculates the availability based on the uptimes of all platforms
Dim Availability As Single
Availability = 0

For i = 1 To nPlatforms
    Availabilities(i) = UpTimes(i) / GetClock()
Next

For i = 1 To nPlatforms
    Availability = Availability + Availabilities(i)
Next

AvgAvailability = AvgAvailability + (Availability / nPlatforms)
```

```
End Function
```

Listing B.3: Components Module

```
Public Const Functioning As Integer = 2
Public Const Degraded As Integer = 1
Public Const Failed As Integer = 0
Public CompNumb As Integer
Public Repairable As Boolean

Type Component
Number As Integer
PlatformNumb As Integer
State As Integer
Repairable As Boolean
CMOrder As Boolean
PMOrder As Boolean
Event As Integer
MTTFd As Single
MTTFcs As Single
MTTFcd As Single
End Type

Public nComponents As Integer
Public Components() As Component

Function InitComponents()
'Loop to initialize all components
Dim numbComp As Integer
Dim totComp As Integer
numbComp = Worksheets("SpareSimulation").Range("nComp").Value
totComp = numbComp * totPlat
nComponents = 0
Repairable = Worksheets("SpareSimulation").Range("Repairable").Value

ReDim Components(totComp)

```

```
For i = 1 To totPlat
    For j = 1 To numbComp
        AddComponent i
    Next j
Next i
End Function

Function AddComponent(ByVal plat As Integer)
'Initialization of one single component
nComponents = nComponents + 1

With Components(nComponents)
    .Number = nComponents
    .PlatformNumb = plat
    .State = Functioning
    .Repairable = Repairable
    .CMOrder = False
    .PMOrder = False
    .MTTFd = 1 / Worksheets("SpareSimulation").Range("LD").Value
    .MTTFcs = 1 / Worksheets("SpareSimulation").Range("LCS").Value
    .MTTFcd = 1 / Worksheets("SpareSimulation").Range("LCD").Value
End With

'Generate failure of this component
GenerateFailureFromState2 nComponents
EventNotice MTTF + GetClock(), CallBackData("OnComponentFailure", nComponents, FC)
End Function
```

Listing B.4: Costs Module

```
Dim TransportCosts As Single
Dim RepairCosts As Single
Dim PMCosts As Single
Dim DowntimeCosts As Single
Dim CMCosts As Single
Dim HoldingCosts As Single
Dim OrderCosts As Single
```

```
Dim OverhaulCosts As Single
Dim CostPerTransfer As Single
Dim CostPerEmergencyTrans As Single
Dim CostPerRepair As Single
Dim CostPerPMReplacement As Single
Dim CostPerPMPeriod As Single
Public DowntimePerT As Single
Dim CostPerCM As Single
Dim HoldingPlatformPerT As Single
Dim HoldingBasePerT As Single
Dim CostPerBaseOrder As Single
Dim OrderCostPerItem As Single
Dim CostPerPlatformOrder As Single
Dim CostPerOverhaul As Single
Dim CBMcosts As Single
Dim CBMpert As Single

Function InitCosts()
'Initialize all costs
CBMcosts = 0
TransportCosts = 0
RepairCosts = 0
PMCosts = 0
DowntimeCosts = 0
CMCosts = 0
HoldingCosts = 0
OrderCosts = 0
OverhaulCosts = 0
CostPerTransfer = Worksheets("SpareSimulation").Range("CostPerTransfer").Value
CostPerEmergencyTrans = Worksheets("SpareSimulation").Range("CostPerEmergencyTrans").
    Value
CostPerRepair = Worksheets("SpareSimulation").Range("CostPerRepair").Value
CostPerPMReplacement = Worksheets("SpareSimulation").Range("CostPerPMReplacement").Value
CostPerPMPeriod = Worksheets("SpareSimulation").Range("CostPerPMPeriod").Value
DowntimePerT = Worksheets("SpareSimulation").Range("DowntimePerT").Value
CostPerCM = Worksheets("SpareSimulation").Range("CostPerCM").Value
```

```
HoldingPlatformPerT = Worksheets("SpareSimulation").Range("HoldingPlatformPerT").Value
HoldingBasePerT = Worksheets("SpareSimulation").Range("HoldingBasePerT").Value
CostPerBaseOrder = Worksheets("SpareSimulation").Range("CostPerBaseOrder").Value
OrderCostPerItem = Worksheets("SpareSimulation").Range("OrderCostPerItem").Value
CostPerPlatformOrder = Worksheets("SpareSimulation").Range("CostPerPlatformOrder").Value
CostPerOverhaul = Worksheets("SpareSimulation").Range("CostPerOverhaul").Value
CBMpert = Worksheets("SpareSimulation").Range("CBMpert").Value
End Function

Function CalcTotCosts()
'Calculate total costs of one simulation run
AverageTotCosts = AverageTotCosts + CBMcosts + TransportCosts + RepairCosts + PMCosts +
    DowntimeCosts + CMCosts + HoldingCosts + OrderCosts + OverhaulCosts
End Function

Function CalcCBMcosts()
Dim x As Single
x = TimeElapsed()

CBMcosts = CBMcosts + x * CBMpert
End Function

Function CalcTransportCosts(Q As Integer)
TransportCosts = TransportCosts + Q * CostPerTransfer
End Function

Function CalcEmergencyTransCosts()
TransportCosts = TransportCosts + CostPerEmergencyTrans
End Function

Function CalcRepairCosts()
RepairCosts = RepairCosts + CostPerRepair
End Function

Function CalcPMReplacementCosts()
PMCosts = PMCosts + CostPerPMReplacement
```

```
End Function

Function CalcPMCosts()
PMCosts = PMCosts + CostPerPMPeriod
End Function

Function CalcDowntimeCosts()
Dim x As Single
x = TimeElapsed()

For i = 1 To nPlatforms
    If Platforms(i).State = Failed Then
    DowntimeCosts = DowntimeCosts + x * DowntimePerT
    ElseIf Platforms(i).State = Degraded Then
    DowntimeCosts = DowntimeCosts + x * DowntimePerT * (1 - DegradedProduction)
    End If
Next
End Function

Function CalcCMCosts()
CMCosts = CMCosts + CostPerCM
End Function

Function CalcHoldingCosts()
Dim x As Single
x = TimeElapsed()

For i = 1 To nPlatforms
    HoldingCosts = HoldingCosts + x * HoldingPlatformPerT * Platforms(i).Stock
Next

HoldingCosts = HoldingCosts + x * HoldingBasePerT * Bases(1).Stock
End Function

Function CalcBaseOrderCosts(Q As Integer)
OrderCosts = OrderCosts + CostPerOrder + Q * OrderCostPerItem
```

```
End Function

Function CalcPlatformOrderCosts()
OrderCosts = OrderCosts + CostPerOrder
End Function

Function CalcOverhaulCosts()
OverhaulCosts = OverhaulCosts + nComponents * CostPerOverhaul
End Function
```

Listing B.5: Failures Module

```
Public FTSpfd As Single
Public FC As Integer
Public MTTF As Single

Function GenerateFailureFromState2(ByVal comp As Integer)
'Generate failure from functioning state
CompNumb = comp
Dim Tcrit As Single
Dim Tdegr As Single

Tcrit = rndExponential(Components(CompNumb).MTTFcs)
Tdegr = rndExponential(Components(CompNumb).MTTFd)

'This determines whether the failure is critical or degraded, depending on which time is
    lower.
If Tdegr < Tcrit Then
    MTTF = Tdegr
    FC = 1
Else
    MTTF = Tcrit
    FC = 0
End If
End Function

Function GenerateFailureFromState1(ByVal comp As Integer)
```

```
'Generate failure from degraded state
CompNumb = comp
Dim newMTTF As Single
newMTTF = 1 / (1 / Components(CompNumb).MTTFcs + 1 / Components(CompNumb).MTTFcd) 'the MTTF to a critical state is now formed by both critical shock as critical degraded
FC = 0
MTTF = rndExponential(newMTTF)
End Function

Function OnComponentFailure(ByVal comp As Integer, ByVal crit As Integer)
CompNumb = comp
PlatNumb = Components(CompNumb).PlatformNumb
FC = crit

If FC = 1 Then
    DegradedFailure CompNumb, PlatNumb
Else
    CriticalFailure CompNumb, PlatNumb
    If Components(CompNumb).Repairable Then
    EventNotice rndExponential(MTTRc) + GetClock() + rndUAB(OrderHandlingTime, OrderHandlingTime + transTime), CallBackData("OnSpareRepair") 'spare repair time = MTTR + how much time it takes to transport it back
    End If
End If
End Function

Function CriticalFailure(ByVal comp As Integer, ByVal plat As Integer)
CompNumb = comp
PlatNumb = plat

CalcCMCosts

Components(CompNumb).State = Failed
Platforms(PlatNumb).State = Failed

If Platforms(PlatNumb).Stock > 0 Then 'if stock > 0 then replace component
```

```
        Platforms(PlatNumb).Stock = Platforms(PlatNumb).Stock - 1
        Components(CompNumb).CMOrder = True
        Components(CompNumb).Event = EventNotice(rndExponential(MDTc) + GetClock(), CallBackData("OnComponentReplacement", CompNumb))
    ElseIf emergTrans Then
        If CheckEmergencyTransfer(CompNumb) Then
            Components(CompNumb).CMOrder = True
        End If
End If

If PM Then
PMuponCriticalFailure CompNumb, PlatNumb
End If
CheckToOrderPlatform PlatNumb
End Function

Function DegradedFailure(ByVal comp As Integer, ByVal plat As Integer)
CompNumb = comp
PlatNumb = plat

Components(CompNumb).State = Degraded
Platforms(PlatNumb).State = Degraded
GenerateFailureFromState1 CompNumb 'create critical failure
EventNotice MTTF + GetClock(), CallBackData("OnComponentFailure", CompNumb, 0)

If PM Then 'if PM we detect the failure during the next test period
EventNotice Platforms(PlatNumb).NextPM + 0.001, CallBackData("OnPMReplacement", CompNumb) '+0.001 in order to make sure replacement is after beginning of PM period
End If

If CBM Then 'if CBM then we detect this degraded failure when it occurs
    If Bases(1).Stock > 0 Then 'if base has enough stock, reserve item
        EventNotice Platforms(PlatNumb).NextPM, CallBackData("OnPlatformOrderArrivalSR", PlatNumb, 1)
        Bases(1).Stock = Bases(1).Stock - 1
        Bases(1).StockReserved = Bases(1).StockReserved + 1
```

```
        CheckToOrderBase
    Else
        QueueAdd PlatNumb, 1, True 'create BO
    End If
Components(comp).PMOrder = True
End If
End Function

Function FailToStart(ByVal comp As Integer, ByVal plat As Integer) As Boolean
CompNumb = comp
PlatNumb = plat

FailToStart = False
Dim x As Single

Randomize
x = Rnd()

If x < FTSpfd Then 'fail to start
    FailToStart = True
    Components(CompNumb).State = Failed
    Platforms(PlatNumb).State = Failed
        If Platforms(PlatNumb).Stock > 0 Then
            Platforms(PlatNumb).Stock = Platforms(PlatNumb).Stock - 1
            Components(CompNumb).CMOrder = True
            Components(CompNumb).Event = EventNotice(rndExponential(MTTRc) + GetClock(),
    CallBackData("OnComponentReplacement", CompNumb))
        End If
    CheckToOrderPlatform PlatNumb
End If
End Function
```

Listing B.6: Platform Module

```
Public PlatNumb As Integer
Public BaseOrderToLvl As Integer
Public BaseOrderLvl As Integer
```

```
Public BaseStockLvl As Integer
Public PlatformOrderToLvl As Integer
Public PlatformOrderLvl As Integer
Public PlatformStockLvl As Integer

Type Platform
Number As Integer
Stock As Integer
OrderToLvl As Integer
OrderLvl As Integer
State As Integer
Ordered As Boolean
NextPM As Single
PMperiod As Boolean
PMcritical As Boolean
End Type
Public totPlat As Integer
Public nPlatforms As Integer
Public Platforms() As Platform
Public OrderToLevel As Integer
Public OrderLevel As Integer

Type Base
Number As Integer
Stock As Integer
OrderToLvl As Integer
OrderLvl As Integer
Ordered As Boolean
StockReserved As Integer
End Type
Public nBases As Integer
Public Bases(1) As Base

Function InitPlatforms()
'Loop to initialize all platforms
nPlatforms = 0
```

```
nBases = 0
totPlat = Worksheets("SpareSimulation").Range("nPlat").Value

ReDim Platforms(totPlat)

PlatformOrderToLvl = Worksheets("SpareSimulation").Range("PLATOTL").Value
PlatformOrderLvl = Worksheets("SpareSimulation").Range("PLATOL").Value
PlatformStockLvl = Worksheets("SpareSimulation").Range("PLATIS").Value
BaseOrderToLvl = Worksheets("SpareSimulation").Range("BASEOTL").Value
BaseOrderLvl = Worksheets("SpareSimulation").Range("BASEOL").Value
BaseStockLvl = Worksheets("SpareSimulation").Range("BASEIS").Value

For i = 1 To totPlat
    AddPlatform
Next

ReDim UpTimes(nPlatforms)
ReDim Availabilities(nPlatforms)
AddBase

End Function

Function AddPlatform()
'Initialization of one platform
nPlatforms = nPlatforms + 1

With Platforms(nPlatforms)
    .Number = nPlatforms
    .Stock = PlatformStockLvl
    .OrderToLvl = PlatformOrderToLvl
    .OrderLvl = PlatformOrderLvl
    .State = Functioning
    .Ordered = False
    .NextPM = PMInterval
    .PMperiod = False
    .PMcritical = False
```

```
End With
End Function

Function AddBase()
'Initialization of the platform
nBases = nBases + 1
With Bases(nBases)
     .Number = nBases
     .Stock = BaseStockLvl
     .OrderToLvl = BaseOrderToLvl
     .OrderLvl = BaseOrderLvl
     .Ordered = False
     .StockReserved = 0
End With
End Function
```

Listing B.7: Repair and Ordering Module

```
Public MTTRd As Single
Public MTTRc As Single
Public MDTd As Single
Public MDTc As Single

Function OnSpareRepair()
'Finishing of repair is similar as arrival of order of one component at base
CalcRepairCosts
OnBaseOrderArrival 1
End Function

Function CheckToOrderPlatform(ByVal plat As Integer)
PlatNumb = plat
Dim Q As Integer
Dim Q2 As Integer
If Platforms(plat).Ordered Then
     Exit Function
End If

```

```
If Platforms(PlatNumb).Stock <= Platforms(PlatNumb).OrderLvl Then
    Q = Platforms(PlatNumb).OrderToLvl - Platforms(PlatNumb).Stock
    If Q <= Bases(1).Stock + Bases(1).StockReserved Then 'Enough at base to supply the full demand
        EventNotice rndUAB(OrderHandlingTime, OrderHandlingTime + transTime) + GetClock, CallBackData("OnPlatformOrderArrival", PlatNumb, Q)
        If Q <= Bases(1).Stock Then
            Bases(1).Stock = Bases(1).Stock - Q
        Else
            Q = Q - Bases(1).Stock
            Bases(1).Stock = 0
            Bases(1).StockReserved = Bases(1).StockReserved - Q
        End If
    ElseIf Bases(1).Stock + Bases(1).StockReserved > 0 Then 'Only part of the platformorder can be met
        Q2 = Bases(1).Stock + Bases(1).StockReserved
        EventNotice rndUAB(OrderHandlingTime, OrderHandlingTime + transTime) + GetClock, CallBackData("OnPlatformOrderArrival", PlatNumb, Q2)
        Q = Q - Q2
        Bases(1).Stock = 0
        Bases(1).StockReserved = 0
        QueueAdd PlatNumb, Q, False
    Else
        QueueAdd PlatNumb, Q 'no spares at base, create BO
    End If
Platforms(PlatNumb).Ordered = True
End If
CheckToOrderBase
End Function

Function CheckToOrderBase()
Dim Q As Integer

If Bases(1).Ordered Or Components(1).Repairable Then 'Check whether order has been sent out or component is repairable, then no order from base
    Exit Function
```

```
End If

If Bases(1).Stock <= Bases(1).OrderLvl Then
    Q = Bases(1).OrderToLvl - Bases(1).Stock
    EventNotice OrderTime + GetClock(), CallBackData("OnBaseOrderArrival", , Q) 'check times
    Bases(1).Ordered = True
End If
End Function

Function OnBaseOrderArrival(ByVal Q As Integer)
Dim Quantity As Integer
Dim ForPM As Boolean
Dim Q2 As Integer
Bases(1).Ordered = False
Bases(1).Stock = Bases(1).Stock + Q

If Not Components(1).Repairable Then
    CalcBaseOrderCosts Q
End If

'Loop handles BO that are in queue
Do Until IsEmpty Or Bases(1).Stock + Bases(1).StockReserved = 0
QueueRemove PlatNumb, Quantity, ForPM
    If Bases(1).Stock + Bases(1).StockReserved >= Quantity And (Not ForPM) Then 'Complete BO is met
        EventNotice rndUAB(OrderHandlingTime, OrderHandlingTime + transTime) + GetClock, CallBackData("OnPlatformOrderArrival", PlatNumb, Quantity)
        If Quantity <= Bases(1).Stock Then
            Bases(1).Stock = Bases(1).Stock - Quantity
        Else
            Quantity = Quantity - Bases(1).Stock
            Bases(1).Stock = 0
            Bases(1).StockReserved = Bases(1).StockReserved - Q
        End If
```

```
    ElseIf Bases(1).Stock + Bases(1).StockReserved > 0 And (Not ForPM) Then 'Only part of
     the BO can be met
        Q2 = Bases(1).Stock + Bases(1).StockReserved
        EventNotice rndUAB(OrderHandlingTime, OrderHandlingTime + transTime) + GetClock,
    CallBackData("OnPlatformOrderArrival", PlatNumb, Q2)
        Bases(1).Stock = 0
        Bases(1).StockReserved = 0
        Quantity = Quantity - Q2
        QueueAdd PlatNumb, Quantity
    ElseIf Bases(1).Stock > 0 And ForPM Then 'If the BO is for PM then create transfer
    for next PM interval
        EventNotice Platforms(PlatNumb).NextPM, CallBackData("OnPlatformOrderArrivalSR",
    PlatNumb, 1)
        Bases(1).Stock = Bases(1).Stock - 1
        Bases(1).StockReserved = Bases(1).StockReserved + 1
    Else
        QueueAdd PlatNumb, Quantity, True
        Exit Do
    End If
Loop

CheckToOrderBase 'Order new components to be sure that stock did not go under s after
    handling BOs
End Function
```

Listing B.8: Replacing Module

```
Public PM As Boolean
Public PMInterval As Single
Public PMDuration As Single
Public NextPM As Single
Public PMstrat As String
Public Overhaul As Boolean
Public OverhaulInt As Single
Public OverhaulDur As Single
Public PMminstock As Integer
Public RepQual As Single
```

```
Public listCurrent As ListItem
Public listPrevious As ListItem
Public CBM As Boolean

Function OnComponentReplacement(ByVal comp As Integer)
CompNumb = comp
PlatNumb = Components(CompNumb).PlatformNumb

With Components(CompNumb)
    .MTTFcd = .MTTFcd * RepQual
    .MTTFd = .MTTFd * RepQual
End With

GenerateFailureFromState2 CompNumb

With Components(CompNumb)
        .CMOrder = False
        .State = Functioning
End With

'Generate new failure for this component (either FTS or "normal" failure)
If Not FailToStart(CompNumb, PlatNumb) Then
EventNotice MTTF + GetClock(), CallBackData("OnComponentFailure", CompNumb, FC)
End If

FixPlatform PlatNumb
End Function

Function OnPMReplacement(ByVal comp As Integer)
CompNumb = comp
PlatNumb = Components(comp).PlatformNumb

If Platforms(PlatNumb).Stock >= PMminstock And Platforms(PlatNumb).Stock > 0 Then
    Platforms(PlatNumb).Stock = Platforms(PlatNumb).Stock - 1
    ExecutePM CompNumb, PlatNumb
    Components(CompNumb).PMOrder = False
```

```
Else 'if there is not enough stock, postpone PM for next interval
    PostponePM CompNumb, PlatNumb
End If
End Function

Function ExecutePM(ByVal comp As Integer, ByVal plat As Integer)
Dim t As Single
CompNumb = comp
PlatNumb = plat

CalcPMReplacementCosts
t = rndExponential(MDTd)

EventNotice t + GetClock(), CallBackData("OnComponentReplacement", CompNumb)
If t > PMDuration Then 'PM will take longer, so end has to be extended
    SearchListForFunction PlatNumb, "OnEndPMPeriod" 'find and delete initial end of PM period
    AddEndPMPeriod GetClock() + t, PlatNumb
End If
CheckToOrderPlatform PlatNumb

SearchListForFunction CompNumb, "OnComponentFailure" 'find and delete the already generate failure of the component

If SearchListForFunction(PlatNumb, "OnPlatformOrderArrivalSR") And Bases(1).StockReserved > 0 Then 'find and delete reserved transfer for component
    Bases(1).StockReserved = Bases(1).StockReserved - 1
    Bases(1).Stock = Bases(1).Stock + 1
End If
End Function

Function PostponePM(ByVal comp As Integer, ByVal plat As Integer)
CompNumb = comp
PlatNumb = plat

```

```
EventNotice Platforms(PlatNumb).NextPM + 0.01, CallBackData("OnPMReplacement", CompNumb)
    '+0.001 in order to make sure replacement is after beginning of PM period

If Components(comp).PMOrder Then 'if order was made before, then don't make a new
    transfer order
    Exit Function
End If

If Bases(1).Stock > 0 Then
    EventNotice Platforms(PlatNumb).NextPM, CallBackData("OnPlatformOrderArrivalSR",
    PlatNumb, 1)
    Bases(1).Stock = Bases(1).Stock - 1
    Bases(1).StockReserved = Bases(1).StockReserved + 1
Else
    QueueAdd PlatNumb, 1, True
End If
Components(comp).PMOrder = True
End Function

Function FixPlatform(ByVal plat As Integer)
PlatNumb = plat

Dim IsFixed As Boolean
IsFixed = True

If Platforms(PlatNumb).PMperiod Then
    Exit Function
End If

For i = 1 To nComponents 'If a component is in a failed state, platform is not repaired
    yet
    If Components(i).PlatformNumb = PlatNumb And Components(i).State = Failed Then
        IsFixed = False
        Exit For
    End If
Next
```

```

If IsFixed Then
    Platforms(PlatNumb).State = Functioning
End If
End Function

Function InitPM()
For i = 1 To nPlatforms
AddStartPMPeriod Platforms(i).NextPM, Platforms(i).Number
AddEndPMPeriod Platforms(i).NextPM + PMDuration, Platforms(i).Number
Next
End Function

Function AddStartPMPeriod(ByVal t As Single, ByVal plat As Integer)
PlatNumb = plat
EventNotice t, CallBackData("OnStartPMPeriod", PlatNumb)
End Function

Function AddEndPMPeriod(ByVal t As Single, ByVal plat As Integer)
EventNotice t, CallBackData("OnEndPMPeriod", PlatNumb)
End Function

Function OnStartPMPeriod(ByVal plat As Integer)
PlatNumb = plat
CalcPMCosts
Platforms(PlatNumb).NextPM = GetClock() + PMInterval + PMDuration

Platforms(PlatNumb).PMcritical = False
Platforms(PlatNumb).PMperiod = True
Platforms(PlatNumb).State = Failed
End Function

Function OnEndPMPeriod(ByVal plat As Integer)
PlatNumb = plat

Platforms(PlatNumb).PMperiod = False
```

```
FixPlatform PlatNumb
'All components have to start up again
For i = 1 To nComponents
    If (Not Components(i).State = Failed) And PlatNumb = Components(i).PlatformNumb Then
        FailToStart i, plat
    End If
Next

AddStartPMPeriod Platforms(PlatNumb).NextPM, PlatNumb
AddEndPMPeriod Platforms(PlatNumb).NextPM + PMDuration, PlatNumb
End Function

Function PMuponCriticalFailure(ByVal comp As Integer, ByVal plat As Integer)
CompNumb = comp
PlatNumb = plat
Dim PMadded As Boolean
PMadded = False

SearchListForFunction PlatNumb, "OnStartPMPeriod"
SearchListForFunction PlatNumb, "OnEndPMPeriod"
'Start pm interval
CalcPMCosts
Platforms(PlatNumb).NextPM = GetClock() + PMInterval + PMDuration
Platforms(PlatNumb).PMperiod = True
Platforms(PlatNumb).State = Failed
Platforms(PlatNumb).PMcritical = True
AddEndPMPeriod GetClock() + PMDuration, PlatNumb

For i = 1 To nComponents
    If Components(i).PlatformNumb = PlatNumb Then
        If SearchListForPM(i) Then
            If listPrevious Is Nothing Then 'delete PM item from list
                Set listHead = listCurrent.NextItem
            Else
                Set listPrevious.NextItem = listCurrent.NextItem
            End If
```

```
                If (Not i = CompNumb) Then
                    OnPMReplacement i
                End If
            End If
        End If
Next

End Function

Function SearchListForPM(ByVal comp As Integer) As Boolean
SearchListForPM = False
PlatNumb = Components(comp).PlatformNumb

Set listPrevious = Nothing
Set listCurrent = listHead

Do Until listCurrent Is Nothing
        If Platforms(PlatNumb).NextPM + 0.1 < listCurrent.t Then
            Exit Do
        End If
        If listCurrent.Data(1) = comp And listCurrent.Data(0) = "OnPMReplacement" Then
            SearchListForPM = True
            Exit Do
        Else
            Set listPrevious = listCurrent
            Set listCurrent = listCurrent.NextItem
        End If
Loop
End Function

Function SearchListForFunction(ByVal x As Integer, ByVal FuncName As String) As Boolean
'Searches any action in the last with relation to x(=comp or plat) and deletes this action
Set listPrevious = Nothing
Set listCurrent = listHead
SearchListForFunction = False
```

```

Do While Not listCurrent Is Nothing
        If listCurrent.Data(1) = x And listCurrent.Data(0) = FuncName Then
            If listPrevious Is Nothing Then 'delete PM item from list
                Set listHead = listCurrent.NextItem
            Else
                Set listPrevious.NextItem = listCurrent.NextItem
            End If
            SearchListForFunction = True
            Exit Do
        Else
            Set listPrevious = listCurrent
            Set listCurrent = listCurrent.NextItem
        End If
Loop
End Function

Function InitOverhaul()
OverhaulInt = Worksheets("SpareSimulation").Range("OVERHAULint").Value
OverhaulDur = Worksheets("SpareSimulation").Range("OVERHAULdur").Value

AddStartOverhaul OverhaulInt
AddEndOverhaul OverhaulInt + OverhaulDur
End Function

Function AddStartOverhaul(ByVal t As Single)
EventNotice t, CallBackData("OnStartOverhaul")
End Function

Function AddEndOverhaul(ByVal t As Single)
EventNotice t, CallBackData("OnEndOverhaul")
End Function

Function OnStartOverhaul()
Dim ToDelete As Boolean
ToDelete = True
```

```
CalcOverhaulCosts

'delete all future failures and planned replacements
Do While ToDelete = True
    DeleteFutureEvents ToDelete
Loop

For i = 1 To nPlatforms
    Platforms(i).State = Failed
Next
End Function

Function OnEndOverhaul()

For i = 1 To nComponents
With Components(i)
    .MTTFd = 1 / Worksheets("SpareSimulation").Range("LD").Value 'reset the lambdas to
    initial values
    .MTTFcs = 1 / Worksheets("SpareSimulation").Range("LCS").Value
    .MTTFcd = 1 / Worksheets("SpareSimulation").Range("LCD").Value
    .State = Functioning
    .CMOrder = False
    .PMOrder = False
End With
Next

For i = 1 To nPlatforms
Platforms(i).State = Functioning
Next

'Generate failures for each component
For i = 1 To nComponents
    plat = Components(i).PlatformNumb
    Components(i).State = Functioning
    If Not (FailToStart(i, plat)) Then
        GenerateFailureFromState2 i
```

```
        EventNotice MTTF + GetClock(), CallBackData("OnComponentFailure", i, FC)
    End If
Next

'If we have PM initialize PM periods again
If PM Then
For i = 1 To nPlatforms
Platforms(i).NextPM = GetClock() + PMInterval
AddStartPMPeriod Platforms(i).NextPM, i
AddEndPMPeriod Platforms(i).NextPM + PMDuration, i
Next
End If

'add next overhaul period
AddStartOverhaul GetClock() + OverhaulInt
AddEndOverhaul GetClock() + OverhaulInt + OverhaulDur
End Function

Function DeleteFutureEvents(ByRef ToDelete As Boolean)
Set listPrevious = Nothing
Set listCurrent = listHead

Do While Not listCurrent Is Nothing
        If listCurrent.Data(0) = "OnComponentFailure" Or listCurrent.Data(0) = "
    OnPMReplacement" Or listCurrent.Data(0) = "OnStartPMPeriod" Or listCurrent.Data(0) =
    "OnEndPMPeriod" Then
            If listPrevious Is Nothing Then
                Set listHead = listCurrent.NextItem
            Else
                Set listPrevious.NextItem = listCurrent.NextItem
            End If
            ToDelete = True
            Exit Do
        ElseIf listCurrent.Data(0) = "OnComponentReplacement" Then
            If listPrevious Is Nothing Then
                Set listHead = listCurrent.NextItem
```

```
                Else
                    Set listPrevious.NextItem = listCurrent.NextItem
                End If
            CompNumb = listCurrent.Data(1)
            PlatNumb = Components(CompNumb).PlatformNumb
            Platforms(PlatNumb).Stock = Platforms(PlatNumb).Stock + 1 'the spare part has not been used, so put it back in stock
            ToDelete = True
            Exit Do
        Else
            Set listPrevious = listCurrent
            Set listCurrent = listCurrent.NextItem
            ToDelete = False
        End If
Loop
End Function
```

Listing B.9: Random Library Module

```
Function rndUAB(A As Single, B As Single)
'       Returns a random number uniformly distributed on (A,B)
'       Even if A > B, this will work as B-A will then be -ve
If (A <> B) Then
    Randomize
    rndUAB = A + ((B - A) * Rnd())
Else
    rndUAB = A
End If
End Function

Function rndExponential(ByVal mu As Single)
'
'         Returns a random number exponentially distributed with
'         mean MU
'
Dim x As Single
x = 0
```

```

If (mu < 0) Then
    MsgBox "Error in rndExponential"
    rndExponential = 0
Else
    Randomize
    Do While x = 0
        x = Rnd()
    Loop
    rndExponential = -Log(x) * mu
End If
End Function
```

Listing B.10: Transfers Module

```
Function CheckEmergencyTransfer(ByVal comp As Integer) As Boolean
'Function checks whether emergency transfer is cheaper than waiting for next supply
Dim CostNormal As Single
Dim CostEmergency As Single
Dim t1 As Single
Dim t2 As Single
CompNumb = comp
PlatNumb = Components(CompNumb).PlatformNumb
CheckEmergencyTransfer = False

If Bases(1).Stock + Bases(1).StockReserved = 0 Then 'if base stock is 0 then we can't
    supply
    Exit Function
End If

If SearchListForOrderArrival(PlatNumb, listCurrent, listPrevious) Then 'check when next
    order arrival is
    t1 = listCurrent.t - GetClock()
    t2 = rndUAB(OrderHandlingTime, OrderHandlingTime + transTime)
    If t1 < transTime Then
        CostNormal = t1 * DowntimePerT + CostPerTransfer
    Else
```

```
            CostNormal = t2 * DowntimePerT + CostPerTransfer
        End If
Else
        CostNormal = t2 * DowntimePerT + CostPerTransfer
End If

CostEmergency = emergTransTime * DowntimePerT + CostPerEmergencyTrans

If CostEmergency < CostNormal Then 'When emergency costs are cheaper, we call for an
    emergency transfer
    EventNotice t2 + GetClock(), CallBackData("OnPlatformEmergencyArrival", CompNumb)
    CheckEmergencyTransfer = True
    If Bases(1).Stock > 0 Then
        Bases(1).Stock = Bases(1).Stock - 1
    Else
        Bases(1).StockReserved = Bases(1).StockReserved - 1
    End If
End If
End Function

Function SearchListForOrderArrival(ByVal plat As Integer, ByRef listCurrent As ListItem,
    ByRef listPrevious As ListItem) As Boolean
SearchListForOrderArrival = False
Set listPrevious = Nothing
Set listCurrent = listHead

Do While Not listCurrent Is Nothing
        If listCurrent.Data(1) = plat And listCurrent.Data(0) = "OnPlatformOrderArrival"
    Then
            SearchListForOrderArrival = True
            Exit Do
        Else
            Set listPrevious = listCurrent
            Set listCurrent = listCurrent.NextItem
        End If
Loop
```

```
End Function

Function OnPlatformEmergencyArrival(ByVal comp As Integer)
CompNumb = comp
CalcEmergencyTransCosts
EventNotice rndExponential(MDTc) + GetClock(), CallBackData("OnComponentReplacement",
    CompNumb)
End Function

Function OnPlatformOrderArrival(ByVal plat As Integer, ByVal Q As Integer)
PlatNumb = plat
Dim i As Integer
i = 1

CalcPlatformOrderCosts
CalcTransportCosts Q

Platforms(PlatNumb).Ordered = False
Platforms(PlatNumb).Stock = Platforms(plat).Stock + Q

'If we still have failed components that are not being repaired yet, a repair order is
    issued
Do While Platforms(PlatNumb).Stock > 0
    If Components(i).PlatformNumb = PlatNumb And Components(i).State = Failed And
    Components(i).CMOrder = False Then
        Components(i).CMOrder = True
        EventNotice rndExponential(MDTc) + GetClock(), CallBackData("
    OnComponentReplacement", Components(i).Number)
        Platforms(PlatNumb).Stock = Platforms(PlatNumb).Stock - 1
    End If
    i = i + 1
    If i > nComponents Then
        Exit Do
    End If
Loop

```

```
CheckToOrderPlatform PlatNumb
End Function

Function OnPlatformOrderArrivalSR(ByVal plat As Integer, ByVal Q As Integer)
Dim Quantity As Integer
Quantity = Q
PlatNumb = plat

If Bases(1).StockReserved > 0 Then
    Bases(1).StockReserved = Bases(1).StockReserved - 1
    OnPlatformOrderArrival PlatNumb, Quantity
ElseIf Bases(1).Stock > 0 Then
    Bases(1).Stock = Bases(1).Stock - 1
    OnPlatformOrderArrival PlatNumb, Quantity
Else
    QueueAdd PlatNumb, Quantity, True
End If
End Function

Public Function QueueAdd(ByVal PlatNumb As Integer, Q As Integer, Optional PM As Boolean
    = False)
    Dim qNew As QueueItem
    Set qNew = New QueueItem

    qNew.PlatformNumb = PlatNumb
    qNew.OrderQuantity = Q
    qNew.ForPM = PM

    ' What if queue is empty? Better point
    ' both the front and rear pointers at the
    ' new item.
    If IsEmpty Then
        Set qFront = qNew
        Set qRear = qNew
    Else
        Set qRear.NextItem = qNew
```

```
        Set qRear = qNew
    End If
End Function

Public Function QueueRemove(ByRef PlatNumb As Integer, ByRef Quantity As Integer, ByRef ForPM As Boolean)
    ' Remove an item from the head of the
    ' list, and return its value.

    If IsEmpty Then
        QueueRemove = Null
    Else
        PlatNumb = qFront.PlatformNumb
        Quantity = qFront.OrderQuantity
        ForPM = qFront.ForPM
        ' If there's only one item
        ' in the queue, qFront and qRear
        ' will be pointing to the same node.
        ' Use the Is operator to test for that.
        If qFront Is qRear Then
            Set qFront = Nothing
            Set qRear = Nothing
        Else
            Set qFront = qFront.NextItem
        End If
    End If
End Function

Property Get IsEmpty() As Boolean
    ' Return True if the queue contains
    ' no items.

    IsEmpty = ((qFront Is Nothing) And (qRear Is Nothing))
End Property
```

Listing B.11: ListItem Class

```
Public t As Single
Public Data As Variant
Public NextItem As ListItem

Private Sub Class_Initialize()
    Set NextItem = Nothing
End Sub

Private Sub Class_Terminate()
    Set NextItem = Nothing
End Sub
```

Listing B.12: QueueItem Class

```
Public NextItem As QueueItem
Public PlatformNumb As Integer
Public OrderQuantity As Integer
Public ForPM As Boolean

Private Sub Class_Initialize()
    Set NextItem = Nothing
End Sub

Private Sub Class_Terminate()
    Set NextItem = Nothing
End Sub
```

B.2 Genetic Algorithm

The following listing contains the GA code that is used for the joint optimization. The code for the optimization of the DES is similar to this, except for the parameters for the PM interval, overhaul interval and CBM. It is therefore not used again. The main code of the tool has to be adapted slightly, as some of the input parameters come from the GA code instead of the worksheet. These changes are not shown here.

Listing B.13: Genetic Algorithm for Joint Optimization

```
Type Individual
BaseOrderToLvl As Integer
BaseOrderLvl As Integer
PlatformOrderToLvl As Integer
PlatformOrderLvl As Integer
PMinterval As Single
OverhaulInt As Single
CBM As Boolean
FitnessLvl As Single
End Type
Dim popL As Integer
Dim genL As Integer
Dim Ppar As Single
Dim Pmut As Single
Dim Pbest As Single
Dim Iterations As Integer
Dim baseMax As Integer
Dim platMax As Integer
Dim pmMax As Single

Public Population() As Individual

Sub MainGA()
Dim indexLow As Integer
Dim indexHigh As Integer
indexLow = 0
indexHigh = 0
InitVar
InitPopulation

For i = 1 To Iterations
    indexLow = lowEvaluation()
    DoCrossover indexLow
    DoMutation indexLow
```

```
    Population(indexLow).FitnessLvl = GetFitnessLvl(Population(indexLow).BaseOrderToLvl,
    Population(indexLow).BaseOrderLvl, Population(indexLow).PlatformOrderToLvl,
    Population(indexLow).PlatformOrderLvl, Population(indexLow).OverhaulInt, Population(
    indexLow).PMinterval, Population(indexLow).CBM) 'calculate the fitnesslvl of the new
    individual by running the DES
    indexLow = lowEvaluation()
    indexHigh = highEvaluation()
    DoBestReplication indexLow, indexHigh
    Worksheets("SpareSimulation").Range("B22").Value = Population(indexHigh).
    BaseOrderToLvl
    Worksheets("SpareSimulation").Range("B23").Value = Population(indexHigh).BaseOrderLvl
    Worksheets("SpareSimulation").Range("B24").Value = Population(indexHigh).
    PlatformOrderToLvl
    Worksheets("SpareSimulation").Range("B25").Value = Population(indexHigh).
    PlatformOrderLvl
    Worksheets("SpareSimulation").Range("B26").Value = Population(indexHigh).OverhaulInt
    Worksheets("SpareSimulation").Range("B27").Value = Population(indexHigh).PMinterval
    Worksheets("SpareSimulation").Range("B28").Value = Population(indexHigh).CBM
    Worksheets("SpareSimulation").Range("B29").Value = Population(indexHigh).FitnessLvl
    ActiveWorkbook.Save
Next

End Sub

Function InitVar()
popL = 100
genL = 4
Ppar = 0.7
Pmut = 0.5
Pbest = 0.3
Iterations = 100
baseMax = 15
platMax = 15
pmMax = 219000
End Function

```

```
Function InitPopulation()
'Initialiaze population
ReDim Population(popL - 1)

For i = 0 To popL - 1
    Population(i).BaseOrderToLvl = GenerateBaseOrderToLvl(1, baseMax)
    Population(i).BaseOrderLvl = GenerateBaseOrderLvl(0, Population(i).BaseOrderToLvl -
    1)
    Population(i).PlatformOrderToLvl = GeneratePlatformOrderToLvl(1, platMax)
    Population(i).PlatformOrderLvl = GeneratePlatformOrderLvl(0, Population(i).
    PlatformOrderToLvl - 1)
    Population(i).OverhaulInt = GenerateOverhaulInt(1000, pmMax)
    Population(i).PMinterval = GeneratePMinterval(1000, Population(i).OverhaulInt)
    Population(i).CBM = GenerateCBM()
    Population(i).FitnessLvl = GetFitnessLvl(Population(i).BaseOrderToLvl, Population(i).
    BaseOrderLvl, Population(i).PlatformOrderToLvl, Population(i).PlatformOrderLvl,
    Population(i).OverhaulInt, Population(i).PMinterval, Population(i).CBM) 'calculate
    the fitnesslvl of the new individual by running the DES
Next
End Function

Function DoCrossover(ByVal indexLow)
Dim x As Integer
Dim y As Integer
Dim p As Single
x = indexLow
y = indexLow

Do While x = indexLow
    x = CInt(rndUAB(0, popL - 1))
Loop
Do While y = indexLow
    y = CInt(rndUAB(0, popL - 1))
Loop

If Population(x).FitnessLvl < Population(y).FitnessLvl Then
```

```
    p = Ppar 'x is fitter
Else
    p = 1 - Ppar 'y is fitter
End If

'Chose stock policy base from one of the parents
Randomize
If Rnd() < p Then
    Population(indexLow).BaseOrderToLvl = Population(x).BaseOrderToLvl
    Population(indexLow).BaseOrderLvl = Population(x).BaseOrderLvl
Else
    Population(indexLow).BaseOrderToLvl = Population(y).BaseOrderToLvl
    Population(indexLow).BaseOrderLvl = Population(y).BaseOrderLvl
End If
Randomize
'Chose stock policy platform from one of the parents
If Rnd() < p Then
    Population(indexLow).PlatformOrderToLvl = Population(x).PlatformOrderToLvl
    Population(indexLow).PlatformOrderLvl = Population(x).PlatformOrderLvl
Else
    Population(indexLow).PlatformOrderToLvl = Population(y).PlatformOrderToLvl
    Population(indexLow).PlatformOrderLvl = Population(y).PlatformOrderLvl
End If
'Chose overhaul interval from one of the parents
Randomize
If Rnd() < p Then
    Population(indexLow).OverhaulInt = Population(x).OverhaulInt
Else
    Population(indexLow).OverhaulInt = Population(y).OverhaulInt
End If
'Chose test interval from one of the parents. Chose from both parents when PM interval of
     both parents is lower than overhaul interval
If Population(x).PMinterval < Population(indexLow).OverhaulInt And Population(y).
    PMinterval < Population(indexLow).OverhaulInt Then
    Randomize
    If Rnd() < p Then
```

```
        Population(indexLow).PMinterval = Population(x).PMinterval
    Else
        Population(indexLow).PMinterval = Population(y).PMinterval
    End If
ElseIf Population(x).PMinterval < Population(indexLow).OverhaulInt Then
        Population(indexLow).PMinterval = Population(x).PMinterval
    Else
        Population(indexLow).PMinterval = Population(y).PMinterval
End If
'Chose CBM from one of the parents
Randomize
If Rnd() < p Then
    Population(indexLow).CBM = Population(x).CBM
Else
    Population(indexLow).CBM = Population(y).CBM
End If
'Reset fitness value
Population(indexLow).FitnessLvl = 0
End Function

Function DoMutation(ByVal indexLow)
Dim x As Single

Randomize
If Rnd() > Pmut Then
    Exit Function 'skip mutation 1-Pmut of the iterations
End If

Randomize
x = Rnd()
If x < 1 / 5 Then
    Population(indexLow).BaseOrderToLvl = GenerateBaseOrderToLvl(Population(indexLow).BaseOrderLvl + 1, baseMax)
    Population(indexLow).BaseOrderLvl = GenerateBaseOrderLvl(0, Population(indexLow).BaseOrderToLvl - 1)
ElseIf x < 2 / 5 Then
```

```
    Population(indexLow).PlatformOrderToLvl = GeneratePlatformOrderToLvl(Population(indexLow).PlatformOrderLvl + 1, baseMax)
    Population(indexLow).PlatformOrderLvl = GeneratePlatformOrderLvl(0, Population(indexLow).PlatformOrderToLvl - 1)
ElseIf x < 3 / 5 Then
    Population(indexLow).OverhaulInt = GenerateOverhaulInt(Population(indexLow).PMinterval, pmMax)
ElseIf x < 4 / 5 Then
    Population(indexLow).PMinterval = GeneratePMinterval(0, Population(indexLow).OverhaulInt)
Else
    Population(indexLow).CBM = GenerateCBM()
End If
End Function

Function DoBestReplication(ByVal indexLow As Integer, ByVal indexHigh As Integer)
Randomize
If Rnd() > Pbest Then
    Exit Function 'skip best replication 1-Pbest of the iterations
End If

Population(indexLow).BaseOrderToLvl = Population(indexHigh).BaseOrderToLvl
Population(indexLow).BaseOrderLvl = Population(indexHigh).BaseOrderLvl
Population(indexLow).PlatformOrderToLvl = Population(indexHigh).PlatformOrderToLvl
Population(indexLow).PlatformOrderLvl = Population(indexHigh).PlatformOrderLvl
Population(indexLow).OverhaulInt = Population(indexHigh).OverhaulInt
Population(indexLow).PMinterval = Population(indexHigh).PMinterval
Population(indexLow).CBM = Population(indexHigh).CBM
Population(indexLow).FitnessLvl = Population(indexHigh).FitnessLvl
End Function

Function lowEvaluation() As Integer
'Find the individual with the best fitness (=highest costs)
Dim low As Integer
low = 0
For i = 1 To popL - 1
```

```
    If Population(i).FitnessLvl > Population(low).FitnessLvl Then
    low = i
    End If
Next
lowEvaluation = low
End Function

Function highEvaluation() As Integer
'Find the individual with the best fitness (=lowest costs)
Dim high As Integer
high = 0
For i = 1 To popL - 1
    If Population(i).FitnessLvl < Population(high).FitnessLvl Then
    high = i
    End If
Next
highEvaluation = high
End Function

Function GenerateBaseOrderToLvl(ByVal Min As Integer, ByVal Max As Integer) As Integer
GenerateBaseOrderToLvl = CInt(rndUAB(Min, Max))
End Function

Function GeneratePlatformOrderToLvl(ByVal Min As Integer, ByVal Max As Integer) As
    Integer
GeneratePlatformOrderToLvl = CInt(rndUAB(Min, Max))
End Function

Function GenerateBaseOrderLvl(ByVal Min As Integer, ByVal Max As Integer) As Integer
GenerateBaseOrderLvl = CInt(rndUAB(Min, Max))
End Function

Function GeneratePlatformOrderLvl(ByVal Min As Integer, ByVal Max As Integer) As Integer
GeneratePlatformOrderLvl = CInt(rndUAB(Min, Max))
End Function
```

```
Function GeneratePMinterval(ByVal Min As Single, ByVal Max As Single) As Single
GeneratePMinterval = (Round(rndUAB(Min, Max) / 100)) * 100
End Function

Function GenerateOverhaulInt(ByVal Min As Single, ByVal Max As Single) As Single
GenerateOverhaulInt = (Round(rndUAB(Min, Max) / 100)) * 100
End Function

Function GenerateCBM() As Boolean
Randomize
If Rnd() < 0.5 Then
    GenerateCBM = True
Else
    GenerateCBM = False
End If
End Function
```

Bibliography

Alrabghi, A., Tiwari, A., and Alabdulkarim, A. (2013). Simulation based optimization of joint maintenance and inventory for multi-components manufacturing systems. In *Simulation Conference (WSC), 2013 Winter*.

Arena (2014). https://www.arenasimulation.com/.

Barlow, R. E. and Proschan, F. (1964). Comparison of replacement policies, and renewal theory implications. *The Annals of Mathematical Statistics*.

de Smidt-Destombes, K. S., van der Heijden, M. C., and van Harten, A. (2009). Joint optimisation of spare part inventory, maintenance frequency and repair capacity for k-out-of-n systems. *International Journal of Production Economics*, 118(1):260 – 268.

Getz, K. and Gilbert, M. (2000). *VBA developer's handbook*. SYBEX.

Hokstad, P. and Frøvig, A. (1996). The modelling of degraded and critical failures for components with dormant failures. *Reliability Engineering and System Safety*, 51(2):189–199. cited By 11.

Jardine, A. K., Lin, D., and Banjevic, D. (2006). A review on machinery diagnostics and prognostics implementing condition-based maintenance. *Mechanical Systems and Signal Processing*, 20(7):1483 – 1510.

Jensen, K. and Kristensen, L. M. (2009). *Coloured Petri Nets*. Springer.

Law, A. (2007). *Simulation Modeling and Analysis*. McGraw-Hill, New York.

Murata, T. (1989). Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

Nance, R. E. (1993). A history of discrete event simulation programming languages. Technical report, Virginia Polytechnic Institute and State University.

Paul, R. and Chanev, T. (1997). Optimising a complex discrete event simulation model using a genetic algorithm. *Neural Computing and Applications*, 6(4):229–237. cited By 9.

Robinson, S. (2014). *Discrete-event simulation: A primer*, chapter 2, pages 10–25. John Wiley & Sons Ltd.

Ross, S. M. (2014). *Introduction to Probability Models*. Academic Press.

Sherbrooke, C. (2008). *Optimal Inventory Modeling of Systems: Multi-Echelon Techniques*. Kluwer Academic Publishers, Boston.

SINTEF (2009). *OREDA: Offshore Reliability Data*. OREDA Participants.

van der Aalst, W. and Stahl, C. (2011). *Modeling Business Processes: A Petri Net-Oriented Approach*. The MIT Press.

Van Horenbeek, A., Buré, J., Cattrysse, D., Pintelon, L., and Vansteenwegen, P. (2013). Joint maintenance and inventory optimization systems: A review. *International Journal of Production Economics*, 143(2):499 – 508. Focusing on Inventories: Research and Applications.

Vatn, J. (2012). Introduction to discrete-event simulation.

Wang, L., Chu, J., and Mao, W. (2008). A condition-based order-replacement policy for a single-unit system. *Applied Mathematical Modelling*, 32(11):2274–2289.

Wells, L. (2002). *Performance Analysis using Coloured Petri Nets*. PhD thesis, University of Aarhus.

Westergaard, M. and Verbeek, H. (2011). Efficient implementation of prioritized transitions for high-level petri nets. *Petri Nets and Software Engineering*.

Winston, W. L. (2000). *Simulation Modeling Using @RISK*. Duxbury Press.

Yu, X. and Gen, M. (2010). *Introduction to Evolutionary Algorithms*. Springer London.