Wave Energy Scotland

Operations and Maintenance Simulation Tool

Functionality Report

This report details the purpose and functionality of an Excel-based tool for simulating the operations and maintenance (O&M) phase of wave energy arrays. The tool is intended to provide developers of wave energy technology with the framework to estimate operational expenditure and undertake sensitivity studies into the design of their devices and O&M strategies. Every aspect of the code which forms the basis of the tool is described in detail. An index is provided from page 155 as a form of reference guide for the document. The work has stemmed from an Engineering Doctorate sponsored by Wave Energy Scotland in partnership with the Industrial Doctoral Centre for Offshore Renewable Energy.

Related documents

- *WES 2017. Operations and Maintenance Simulation Tool. MS Excel (Macro-Enabled)*
- *WES 2017. Operations and Maintenance Simulation Tool - Weather Simulation Report. PDF*
- *WES 2017. Operations and Maintenance Simulation Tool - User Guide. PDF*
- *WES 2017. Operations and Maintenance Simulation Tool - Future Upgrades. PDF*

Revision 1

CONTENTS

1 INTRODUCTION

The seas around Scotland are some of the most powerful and inhospitable on the planet, which makes them ideal for deploying wave energy converters (WECs) – devices that use wave action to generate electricity. Wave Energy Scotland (WES) was set up by the Scottish Government in 2014 to fund and support innovative solutions to the technical challenges of harnessing energy from the waves. One aspect making the commercialisation of wave energy devices difficult is the uncertainty surrounding lifetime costs of wave energy arrays, particularly during the operations and maintenance (O&M) phase. Having a reliable means of estimating these costs in as realistic a way as possible is therefore hugely important in the development of the wave energy sector.

Wave Energy Scotland has released an O&M simulation tool to analyse the lifetime logistics of a wave energy array. The tool originates from a research project sponsored by WES through the Industrial Doctoral Centre for Offshore Renewable Energy (www.idcore.ac.uk). The tool was initially developed in collaboration with Pelamis Wave Power, one of the world's leading wave energy technology companies at the time, with an emphasis on commercial-scale WECs rated up to 1MW. The tool's methodology was then applied to the much smaller off-grid WECs being designed by Albatern, another Scottish wave energy developer. These collaborations have enabled WES to produce an O&M simulation tool which can be applied to an array of any type of wave energy converter.

An O&M simulation model is an extremely useful tool for three primary reasons:

1. At early stages of development, an O&M model can help identify critical components which would have the biggest impact on array performance, thus providing feedback into the device design
2. The tool provides estimates of array availability, revenue and operational expenditure which helps to refine Levelised Cost of Energy (LCOE) calculations
3. As device development moves towards real sea testing, the tool can assist in planning aspects of the O&M strategy for future arrays

The tool has been created using Microsoft Excel and the associated VBA programming language. It uses the Monte Carlo method to simulate the occurrence of faults on each WEC in an array by utilising failure rate data. All the components of the device are represented by fault categories, assigned following a Failure Modes and Effects Analysis (FMEA) of the device. The user can choose whether certain faults can be repaired whilst the WEC is offshore, or if all faults require the device to be towed to the safety of a sheltered quayside or onshore O&M base for repair. This 'reactive' maintenance modelling is coupled with the option to include modelling of 'proactive' routine servicing of WECs. Maintenance parameters, such as repair times and parts costs, are defined by the user, as are other aspects of the O&M strategy, such as weather limits for marine operations. The model utilises a time series of weather conditions in order to assess windows of accessibility and calculate revenue generated by the array at each time step. The model simulates the array lifetime as realistically as possible by enforcing logistical constraints, including technician availability and quayside access. A full breakdown (per device and per year) of outputs including availability, revenue and operational expenditure is presented, as well as a table attributing costs to each fault category.

This report details the functionality of the O&M simulation tool and is intended to be a reference guide for model validation, as well as future modifications and enhancements.

2 TERMINOLOGY

Acronyms, abbreviations and potentially unfamiliar words are listed here for user reference.

CBA	- Cost-Benefit Analysis
FMEA	- Failure Modes and Effects Analysis
Hs	- Significant wave height
IDCORE	- Industrial Doctoral Centre for Offshore Renewable Energy
LCOE	- Levelised Cost of Energy
Multicat	- a multipurpose vessel used for marine operations
NaN	- Not a Number
O&M	- Operations and Maintenance
Offsite	- not at the wave energy array site, i.e. onshore or at quayside
Onsite	- at the wave energy array, i.e. offshore
OOP	- Object-Oriented Programming
OPEX	- Operational Expenditure
PTO	- Power Take-Off
Te	- Wave energy period
Tp	- Wave peak period
Transit (in)	- when travelling to and from the wave energy site, with or without towing a WEC
U	- Wind speed
U10	- Wind speed at 10 metres above ground level
VBA	- Visual Basic for Applications (Microsoft Excel's programming language)
WEC	- Wave Energy Converter
WES	- Wave Energy Scotland
Weather window	- a period where weather conditions remain accessible for marine operations

2.1 VBA TERMINOLOGY

Argument	- a variable sent to a procedure for use
Call	- a procedure is 'called' by another one in order for it to undertake its action
Cells	- an in-built VBA function used to refer to an Excel cell
Class module	- secondary object in VBA programming
Const	- a term used to assign a variable a certain value at the beginning of an object or procedure
Data type	- format in which data is stored in a variable
Dim	- a term used to define a variable as a particular data type
Function	- a procedure that performs an action and can return values
Interval	- one time step of the model
Module	- primary object in VBA programming
Procedure	- a set of programming instructions to perform some action
Subroutine	- a procedure that performs an action
Variable	- a temporary holder of information. Data can be in numerous types, such as String (words), Integers (whole number between -32,768 and 32,767), Long (whole number up to 2 billion) or Double (a decimal number)
Worksheet	- VBA term for an Excel spreadsheet

3 MODEL STRUCTURE

The O&M model operates by taking information stored in Excel spreadsheets and processing the data in Visual Basic for Applications (VBA). The outputs are then printed to Excel spreadsheets.

The model interface consists of a primary 'Inputs' spreadsheet with tables detailing potential failures and scheduled maintenance tasks, as well as other spreadsheets for aspects such as workforce arrangements and vessels available for the array. These parameters allow the array to be defined and are used to constrain operations, thereby dictating the O&M strategy.

The processing methodology of the model occurs in VBA and is of an Object-Oriented Programming (OOP) nature. This allows the model to undertake a series of processes at every time step for each year of the array lifetime. Failure rate data is used to simulate the occurrence of faults on each device in the array, and marine operations are carried out to allow repairs to take place. Output data is calculated throughout the model processing and the information is printed in a clear way on spreadsheets. A flowchart of the model structure can be seen in Figure 3.1.

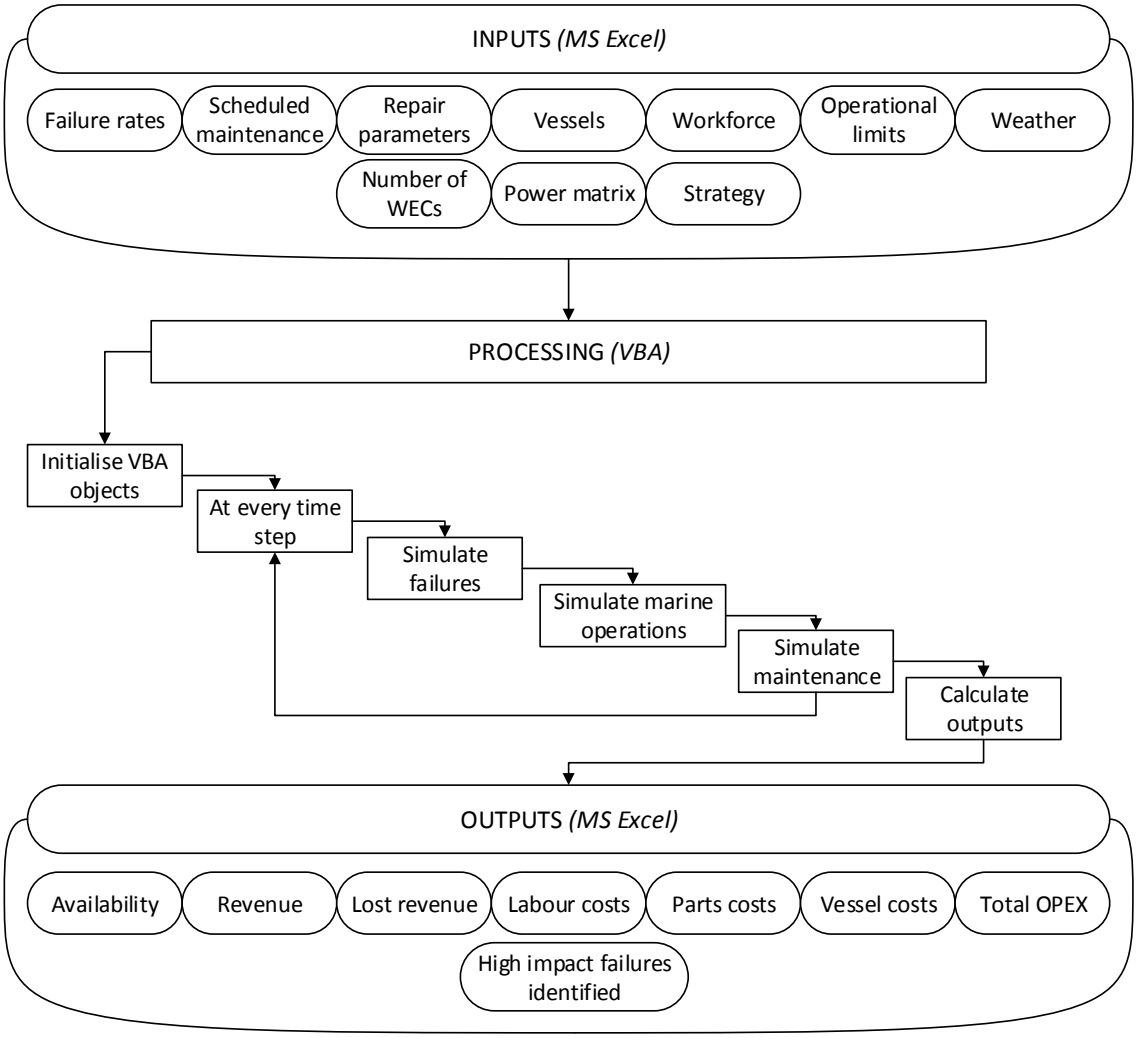

Figure 3.1. O&M model structure (high-level) (Gray 2017)

4 MODEL INPUTS

There are eight spreadsheets that require input from the user. The information contained within these sheets is used by the VBA processing to simulate the lifetime logistics of the defined wave energy array. The primary input spreadsheet is named 'Inputs'. The seven other user-controlled sheets are located to the left of 'Inputs'. After the model has finished a simulation, the results are printed to spreadsheets placed to the right of 'Inputs'. It is important to note that the VBA code reads directly from the input spreadsheets. Therefore, if the value or positioning of input data is modified, then the code may need to be changed accordingly. The eight input sheets are:

- Inputs
- Vessels
- Labour
- Ops Limits
- Power
- Daylight
- Weather
- Hindcast

This section addresses each spreadsheet in turn and details the information contained within.

4.1 INPUTS SHEET

The 'Inputs' spreadsheet contains the primary data required by the O&M model. This includes defining the number of WECs in the array, the project lifetime and the installation details of a WEC. These universal inputs are accompanied by tables detailing the potential faults and scheduled maintenance tasks for WECs or for the entire array. The components and subsystems of the WEC and the array are represented by fault categories. Each category is assigned a failure rate, power loss, vessel required and other parameters defining the repair action. Such parameters are also assigned to the scheduled maintenance events. The spreadsheet also contains Macro buttons for running the processes of the O&M model.

4.1.1 Universal Inputs

The universal inputs are contained within a single column in the 'Inputs' spreadsheet. These are:

- Number of WECs in the array
- Number of failure types
- Number of scheduled maintenance categories
- Array lifetime
- Operational limits for WEC installation
- Vessel required for WEC installation
- Time required onsite for WEC installation
- Number of technicians required for WEC installation
- Number of WECs allowed at O&M base
- Number of WECs allowed at O&M base solely for maintenance
- Number of spare PTO units
- Number of spare instrumentation boxes
- Delivery time for spares

- Cost-benefit analysis for WEC retrieval
- Cost-benefit analysis for onsite WEC repair
- Days allowance until maintenance for cost-benefit analysis
- Marine operations allowed at night
- Location of array
- Use current weather dataset
- Choose a specific weather dataset
- Format of monetary outputs

Many of these inputs need to be added manually by the user. This includes the number of WECs, the array lifetime and the number of scheduled maintenance tasks. The number of failures types is updated automatically as the maximum value of the IDs located in column A. This assumes that there will always be more fault categories than maintenance categories.

The model starts by assuming all WECs have already been installed at the offshore site. However, one possible maintenance strategy is removing faulty WECs (or ones that require routine maintenance) for repair and inspection at the safety of a sheltered harbour or O&M base. As a result, WEC installation details are required by the user in the universal inputs. The operational limits for WEC installation are given as 'types' corresponding to those found in the 'Ops Limit' sheet (section 4.4, page 23). A type can be solely defined by significant wave height, but can also include a wind speed constraint or it can vary depending on wave period. An error message will be displayed if an invalid entry is attempted. The vessel required for WEC installation must be selected from a dropdown list, corresponding to the vessels listed in the 'Vessels' spreadsheet (section 4.2, page 22). The number of hours to undertake a WEC installation does not include vessel transit, as it is the time from when the WEC is towed to the site, until it is ready to start operating. This value can be given as a decimal number to represent fractions of an hour if required (i.e. enter 1.5 if the installation takes one hour and thirty minutes). The number of technicians required for a WEC installation operation must be given as an integer.

The amount of onshore or quayside space is taken into account using the two cells defining the number of WECs allowed at the O&M base. The 'no. WECs allowed at base' value places a constraint of the number of WECs that can be moored at the quayside (or placed onshore, depending on O&M base layout) at any given time. The 'no. WECs allowed at base for maintenance' value can be used to define the amount of space that can be used solely for scheduled maintenance activities. This enables analysis of the strategy of keeping one or more spaces available for emergency WEC repair during periods of high maintenance activity.

The model allows failures to be specified as requiring onsite repair only. This may be relevant for WECs that have a modular design, enabling certain parts to be replaced quickly and safely whilst the device is still offshore. The two examples of replacement parts given in this model are power take-off (PTO) units and instrumentation boxes. Two cells in the universal inputs allow the user to define the number of each of these parts stored at the O&M base. The 'delivery time' value is provided by the user, and is the number of days between using one of the spare parts from the O&M base and a new one being delivered for future use.

Two options are available to the user for implementing a cost-benefit analysis (CBA) to the decision making of the model functionality. If 'retrieval cost-benefit analysis' is set to 'Yes' then the costs of repairing a faulty WEC that requires retrieval (i.e. being taken to the O&M base for repair) are

weighed up against the income generated by the faulty device before a decision is made on undertaking the marine operation. If the value is set to 'No' then a faulty WEC requiring retrieval is repaired as soon as possible. Likewise, if 'onsite repairs cost-benefit analysis' is set to 'Yes' then the CBA is used for faulty WECs that can be repaired offshore. Otherwise, such repairs are undertaken as soon as possible. The 'CBA days allowance for maintenance' value is used by the cost-benefit analysis (if either of the CBA options has been set to 'Yes'). At any given time step, if any scheduled maintenance on the WEC will be due within the specified number of days, then the marine operation is delayed until then.

The user can specify if marine operations can take place at night or if they are constrained to daylight hours. If the 'night operations?' cell is set to 'No' then the daylight hours matrix found in the 'Daylight' spreadsheet (section 4.6, page 25) is used to defined weather windows, in addition to the operational limits. The daylight hours matrix must correspond to the site entered in the 'Array location' cell. This entry is also used to identify which weather dataset is to be used for the model simulations.

The O&M model allows the user to choose which dataset of weather conditions they want to use for simulations. The 'use current weather?' cell also contains the name of the dataset currently located in the 'Weather' spreadsheet. If the value is set to 'Yes' then this dataset is used for the simulations. If the value is set to 'No' then the Excel workbook containing the weather datasets corresponding to the defined site and array lifetime needs to be opened. More information on the creation of these dataset stores can be found in the 'Weather Simulation Report' (WES, 2017a). If 'choose specific dataset?' is set to 'Yes' then a dropdown list appears with the names of all the datasets within the store. If it is set to 'No' then a dataset is chosen randomly from the store.

The user can select whether to show monetary outputs in pounds, thousands of pounds or millions of pounds using the 'output format' entry.

4.1.2 Fault categories

The dominant aspect of the 'Inputs' spreadsheet is the table of fault categories. These categories represent all the components and subsystems within the WEC. In addition, the categories can involve array components, such as subsea central electrical connections, if required by the user. The categories hold a large amount of descriptive and numerical information, most of which is used by the model's VBA code. The likelihood, severity and consequence of all component failures should be considered. This leads to the classification of failures as either 'major, 'intermediate' or 'minor' faults. They can then be separated further into the type of engineering involved, such as 'hydraulic' or 'structural'. This method of classification in the O&M model means that the best way of collating the input data is through a Failure Modes and Effects Analysis (FMEA). This is a well understood procedure, with many guidance documents available in the public domain (e.g. DNV, 2012). An explanation of how to use information from an FMEA to obtain inputs for the O&M model can be seen in Appendix A (page 168). By grouping components into fault categories, the O&M model can run simulations much faster than if it had to operate for every component in the WEC and array. This also allows changes to the WEC design to be analysed and compared in a rapid and efficient manner. The drawback of this method is that there is variability in the inputs to each category because the values are averages of all the components contained within that category. Therefore, a trade-off between the level of variability in the inputs and the speed and adaptability

of the model needs to be made. The names and input values of fault categories will vary for different types of WEC. However, the following fifteen categories provide guidance:

- Major mooring
- Major structure
- Major hydraulic
- Major electrical
- Major communications
- Intermediate mooring
- Intermediate structure
- Intermediate hydraulic
- Intermediate electrical
- Intermediate communications
- Minor mooring
- Minor structure
- Minor hydraulic
- Minor electrical
- Minor communications

The cell containing the ID of each fault category is formatted to be a colour corresponding to its classification. The VBA code reads this colour in order to determine the severity of the failure. For each fault category, the table on the 'Inputs' spreadsheet offers a series of descriptive cells which are not incorporated into the VBA code. Instead, these are used to outline the components and failure modes contained within each category, and provide justification for the input values given. Descriptive entries for each fault category could include:

- Example of components or failure modes
- Consequences of failure
- Knock on effect of failure
- Effect on power capture
- Remote indication
- External visual indication
- Basis of probability

The 'knock on effect' entry has no impact on the model at present. However, it could become possible to modify the failure rates if certain faults has significant knock on effects on the likelihood of other faults, if such information was available.

The 'effect on power capture' entry provides a description for the value provided for 'power loss'. This numerical value indicates the power loss incurred on the entire array due to the occurrence of the given fault category. It needs to be a value between 0 (indicating no power loss) and 1 (shutdown of the entire array). Therefore, if a fault category refers to WEC components, then the Excel formula for that cell should include the number of WECs in the farm:

$$Array\ power\ loss = \frac{WEC\ power\ loss}{Number\ of\ WECs\ in\ array}$$

The 'relevance' entry in the table of fault categories has a dropdown list and indicates whether those components relate to the WEC or the array. This is used by the model to undertake the Monte Carlo simulation of failures, whereby array failures are assessed only once at each time

step, in contrast to WEC failures which are assessed for the number of WECs in the array at each time step. The 'relevance' of a fault category also has major implications on the functionality of the simulated logistics over the course of the array lifetime. The other inputs of each fault category listed below must be applicable to the 'relevance':

- Probability of failure (per year)
- Action required
- Vessel required
- Time required offshore (hours)
- Ops limits type
- Time required onshore (days)
- Parts cost (£)
- Other costs (£)
- Technicians required

The likelihood aspect of the FMEA process means that failure rates of each component in the WEC and array can either be found using reliability handbooks (e.g. OREDA, 2015) or accelerated testing, or can be assumed in the initial instance. In grouping components into fault categories, these individual failure rates can be averaged to produce the annual probabilities of failure entered into the O&M model. The VBA code uses these probabilities to run the Monte Carlo analysis which simulates the occurrence of faults at each time step.

If any fault categories have 'relevance' set to 'array', then it may be likely that the failure rate is dependent on the number of WECs in the array. For example, the number of mooring lines required may not have a linear relationship with the number of WECs. If the model is tailored in this manner, then it may be useful to add a new input spreadsheet that sends information to the 'probability of failure' entry of the appropriate fault category.

The 'action required' must be selected from a dropdown list with the options: 'retrieve WEC', 'replace PTO unit', 'replace instrumentation box' and 'moorings/subsea work'. These refer to the action that is required when that fault category has suffered a failure and is set to be repaired. The 'retrieve WEC' option means that the repair can only be carried out if the WEC is removed from site and taken to the safety of a sheltered quayside or onshore O&M base. Repairs can be undertaken at the offshore site if either of the two 'replace' options are selected, or if 'moorings/subsea work' is required. The 'action required' must be compatible with the 'relevance' (i.e. WEC or array) of the fault category.

The 'vessel required' for the action must be selected from a dropdown list corresponding to the available vessels listed in the 'Vessels' spreadsheet.

The 'time required offshore' must be given in hours, and can be a decimal number (i.e. 0.5 for half an hour). If the selected action is 'retrieve WEC' then this entry refers to the time it takes to disconnect the WEC once the vessel has arrived at site. In other words, it does not include the transit time neither with nor without towing the device. Transit time is also not included if any of the other actions are required, however, it is likely these will take longer than WEC disconnection.

As with the installation operational limits described in the universal inputs section (section 4.1.1), the 'ops limits type' entry in the fault categories refers to a type detailed in the 'Ops Limit's spreadsheet.

For any action apart from 'retrieve WEC', there will be no 'time required onshore'. The entry should therefore read 'N/A'. However, for 'retrieve WEC' actions, this entry refers to the number of days required to repair the WEC at the sheltered quayside or onshore O&M base. This time must take working hours of technicians into account.

The cost of parts is defined following the FMEA process as described in Appendix A (page 168). The 'parts cost' for each fault category is an average of the cost of the components within that category. 'Other costs' can incorporate aspects such as the use of divers or extra equipment (e.g. a drydock), as well as unforeseen costs. Both these entries need to be given with the value in pounds sterling.

If a fault category requires the action 'retrieve WEC' then the 'technicians required' entry refers to the number of personnel required for the repair task at the O&M base for the duration of the 'time required onshore'. The number of technicians required for the actual retrieval operation is obtained from the universal inputs 'install technicians required' entry. For the actions 'replace PTO unit', 'replace instrumentation box' and 'moorings/subsea work', the 'technicians required' entry refers to the number of technicians needed on board the vessel in order to complete the task. This does not account for external personnel such as divers (these are included in 'other costs'). The time required to complete tasks needs to consider technicians' working hours.

4.1.3 Scheduled maintenance

The table of scheduled maintenance tasks in located directly below the table of fault categories. At present, the names of the tasks can be selected from a dropdown list containing three options; 'routine service', 'major components refit' and 'moorings inspection'. The 'relevance' entry must either be defined as 'WEC' or 'Array', as with the fault categories. The relevance must correspond to the type of maintenance:

- Routine service - WEC
- Major components refit - WEC
- Moorings inspection - Array

It should be noted that the cost-benefit analysis aspect of the O&M model will not operate correctly if there are no WEC-related scheduled maintenance tasks defined. The CBA uses these events to calculate the income generated by a WEC up to the point when it is next scheduled for maintenance. This is fundamental to the functionality of the cost-benefit analysis. However, if no WEC-related maintenance tasks are defined, then the O&M model will still operate correctly so long as no CBA option is selected.

More scheduled maintenance tasks can be added to the model, but the VBA code must be changed accordingly (see section 4.1.3). The 'tasks' entry allows the user to note down examples of the jobs to be undertaken in each maintenance event. This information is not used by the VBA code. All the other entries in the table of scheduled maintenance are used by the model:

- Maintenance interval – *'carry out every…(years)'*
- Staggered
- Time of year
- Action required
- Vessel required
- Time required offshore

- Operational limits type
- Time required onshore
- Parts cost
- Other costs
- Inspection costs
- Technicians required

The interval between maintenance events is governed by the 'carry out every…' entry. The value must be given in years as an integer. If the value is set to 1, then that task is carried out every year (and for every WEC if the task's 'relevance' is 'WEC'). If WEC-based maintenance tasks are not to be carried every year, then the entry 'staggered?' is used. For example, if 'routine service' is to be undertaken every two years, and staggered maintenance is chosen, then half the WECs in the array will be serviced in year 1, then year 3, then year 5 etc., whilst the other half will be serviced in even numbered years. The staggered maintenance entry is ignored if the interval is significantly large. For example, the 'major components refit' task might only occur halfway through the array lifetime. Therefore, the model ignores the 'staggered' entry to ensure than all WECs undergo this task at the half-life point.

The 'time of year' must be the name of a season selected from a dropdown list. The maintenance task will be set at the start of the specified season in the correct years. These start dates correspond to meteorological seasons:

- Spring - 1st March
- Summer - 1st June
- Autumn - 1st September
- Winter - 1st December

The remaining input entries in the table of scheduled maintenance tasks operate in a similar manner to the fault categories.

The 'action required' is selected from a dropdown list with the options 'retrieve WEC' and 'moorings work'. Replacement of parts whilst the WEC is at site is not considered in the model at present, but it is possible to build this functionality in if required.

The 'vessel required' must be selected from a dropdown list which corresponds to the available vessels listed in the 'Vessels' spreadsheet.

For WEC-based maintenance, the 'time required offshore' is the time for WEC disconnection and preparation for towing back to the O&M base. For array-based tasks, it is the time to undertake the maintenance actions once arrived at site. The 'time required offshore' does not include transit times. It can be given as either an integer, representing whole hours, or as a decimal number (e.g. 1.5 is one and a half hours).

The 'ops limits type' entry must refer to one of the types listed in the 'Ops Limits' spreadsheet.

Array-based maintenance events will have no 'time required onshore' so this value should be set to 'N/A'. WEC-based events, however, will require a number of days at the O&M base. The entry must account for the working hours of technicians and other logistical aspects.

The ‘parts cost’ entry refers to the cost of any replacement parts required during the maintenance task. ‘Other costs’ provides a budget for unforeseen costs incurred during the maintenance event. It can also include aspects such as extra food and fuel required for longer vessel operations. ‘Inspection costs’ were not included in the table of fault categories. These costs refer to additional, foreseen expenses for the maintenance task. This could include, for example, the user of external personnel or divers for ‘moorings work’. All three of these cost-based entries must be given in pounds sterling.

The number of ‘technicians required’ is the number of personnel needed to undertake the maintenance task. For WEC-based maintenance, this is the number of technicians needed for the full ‘time required onshore’. The number of technicians needed for the retrieval operation is governed by the ‘install technicians required’ entry in the table of universal inputs. For array-based maintenance, the ’technicians required’ entry refers only to personnel from the O&M base required for the operation. Expenses for any additional personnel required, such as divers or third-party contractors, should be included in ‘other costs’ and ‘inspection costs’.

4.2 VESSELS SHEET

The ‘Vessels’ spreadsheet contains all the information related to the available vessels for use in the wave energy array. For each vessel, the following information is listed:

- Name
- ID
- Average speed (in knots)
- Time to site, free (in hours)
- Time to site, towing (in hours)
- Fuel cost per hour (in £)
- Personnel capacity
- Daily hire fees (in £)
- Vessel availability (0 to 1)

Above the table containing this information, however, there are a number of input values used in the ‘Vessels’ spreadsheet and in the VBA code. The ‘number of vessels’ is updated automatically using the ‘vessel ID’ values in the main table. The ‘distance to site’ value needs to be entered by the user in kilometres. It is used to calculate the transit times to and from site. The ‘preparation time’ is also used to calculate transit times. It can account for any tasks that are required before undertaking any marine operation (e.g. safety checks) and must be given in hours (integers or decimal numbers are valid). For user reference, there is value of 1.852 located in a cell which is used to convert vessel speed from knots to kilometres per hour. The final entry outside the main vessels table is the assumed speed of a vessel whilst towing a WEC, provided in knots.

The names of the vessels are used by the ‘Inputs’ spreadsheet to create the dropdown lists which allow the user to select the ‘vessel required’. The ID of the vessels should be listed in ascending order, starting at 1, with no duplicates. The ‘average speed’ must be given in knots and represents the average speed at which the vessel will travel to the offshore site from the O&M base without towing a WEC.

Using the information provided, the 'time to site, free' (i.e. without towing a WEC) and 'time to site, towing' entries are calculated and rounded up to the nearest 15 minutes. If the vessel cannot tow a WEC then the 'time to site, towing' entry should be changed to 'N/A'.

The 'fuel cost per hour' is an estimate of the amount of fuel used by a vessel and must be provided in pounds sterling. Estimates should be obtained using expected fuel consumption of the vessel as well as the most appropriate fuel cost in the array location. Personnel capacity refers to the number of technicians that can be on board the vessel for a marine operation. This does not include the addition of external personnel, such as divers, because it is assumed that the selected vessel is appropriate for the task.

The 'daily hire fees' entry allows the user to select a day rental rate (in pounds sterling) imposed by the vessel operator. This could include vessel crew costs if the 'install technicians required' (defined in the universal inputs section of the 'Inputs' spreadsheet) are not responsible for operating the boat. It should be noted that the 'daily hire fees' are incurred for any amount of a full 24 hour day. In other words, even if the vessel is only used for 2 hours of the day, the full day rate is still charged. If the vessel is owned by the array operator then 'daily hire fees' can be set to zero. In this case, the 'vessel availability' should also be set 1, meaning that the vessel is always available for marine operations on this array. Unless a long term lease agreement is reached, it is likely that a vessel on hire will be available for other projects in addition to the wave energy array. Therefore, 'vessel availability' can be set to a value between 0 and 1 to account for periods when a vessel may be unavailable when required.

4.3 LABOUR SHEET

The 'Labour' spreadsheet contains information about the workforce arrangements and costs at the O&M base. It is assumed that all personnel listed here are capable of undertaking all repairs and maintenance tasks specified in the 'Inputs' spreadsheet. A 'site manager' can act as a 'technician' in undertaking these tasks. The user can change the number of 'site manager's and 'technician's at the O&M base. The total number of workforce personnel updates automatically and is read by the VBA code. The formula in the 'annual salary' entries for each type of personnel must be modified to represent the annual salaries earned by each member of the team. The total annual salary earned by the members of the O&M team is multiplied by the assumed 'overheads multiplier' to give the annual cost of workforce for the array.

The user must select whether to 'enable short term contractors' using a dropdown list. Contractors are employed as extra technicians and it is assumed that the 'site manager' designates them jobs to which they are suited. If 'enable short term contractors' is set to 'yes' then additional personnel is brought in when the repairs and maintenance tasks would otherwise be delayed due to a lack of available technicians. The 'contractor day rate' must be given in pounds sterling and represents the fee paid for a full 24 hour period. For example, if a contractor is only needed for 6 hours, then only a quarter of the day rate is paid. This aspect needs to be taken into account when specifying the 'contractor day rate'. The 'overheads multiplier' is not used for adjusting the 'contractor day rate'.

4.4 OPERATIONAL LIMITS SHEET

The 'Ops Limits' spreadsheet contains the information defining the different types of operational limits used in the model. The user must define the 'number of ops limits types' and subsequently

complete the table of parameters for each of the chosen types. There is currently space for four types of operational limits, but more can be added if required by following the same format. Each type has an initial header with its type ID. The following row contains the number of parameters used to define that type of operational limits. The user must select this to be an integer between 0 and 3 using a dropdown list. Below the 'parameters considered' entry are five rows used for entering parameter values. The possible entries automatically update depending on which value of 'parameter considered' have been selected, as shown in Table 4.1.

Table 4.1. Headers for each parameter considered in the 'Ops Limits' input spreadsheet

Parameters considered	0	1	2	3
Row 3	N/A	Significant Wave Height (m)	Significant Wave Height (m)	Wind speed (kts)
Row 4	N/A		Wind speed (kts)	Lower maximum Hs (m)
Row 5	N/A			Upper maximum Hs (m)
Row 6	N/A			Lower period point (s)
Row 7	N/A			Upper period point (s)

In Table 4.1, the row ID refers to the position of the header, with row 1 containing the 'Type' ID. Once the user has chosen the number of parameters in each type, they must then select appropriate values for each of the entries. Entries for significant wave height (Hs) and wind speed (U) can be entered as either integers or decimal values. If three parameters are considered then the Hs limit depends on what the wave period is. Note: wave period can be represented in multiple ways, most notable wave energy period (Te) and wave peak period (Tp). The notation of wave peak period must be consistent throughout the model and defined in the 'Power' (section 4.5) and 'Weather' (section 4.7) spreadsheets. Alongside each 'type' table there is a graph providing visual representation of the information. These graphs are not used by the VBA code. An example can be seen in Figure 4.1. The spreadsheet also contains some notes to guide the user in completing the input information.

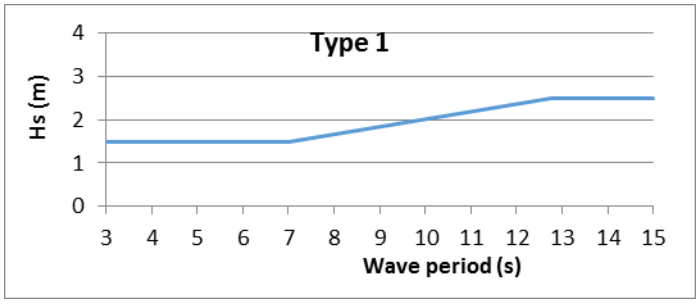

Figure 4.1. Example of an operational limits graph in the 'Ops Limit' spreadsheet

The values on the 'Inputs' spreadsheet use the 'Type' IDs to define the weather limitations for marine operations. The model's VBA code uses the input parameters from the 'Ops Limits' spreadsheet to determine if a weather window is open (i.e. accessible) or closed (i.e. inaccessible). When multiple repairs or maintenance tasks are to be undertaken, the model must choose the

operational limits that are the most restrictive (i.e. the calmest sea and wind conditions). Therefore, it is vital that the operational limit types are listed in order of severity, with the most restrictive limits first. If the severity is not clear (e.g. between a type with 2 parameters and one with 3 parameters), then the hindcast dataset of weather conditions needs to be analysed to find the percentage of 'open' weather windows for each type.

4.5 POWER SHEET

The 'Power' spreadsheet is used to store a power matrix of the WEC (or group of WECs) under analysis in the O&M model. The types of WECs that can be analysed by this model vary. Whilst many will be stand-alone devices deployed alongside others in an array, some possible designs could see WECs joined together directly to increase power output per device. Therefore, there is an entry above the power matrix to allow the user to define the 'number of WECs per power matrix'.

The power values shown in the matrix must be in kilowatts (kW). Values of significant wave height are entered in column A and must be in metres. Values of wave period (this can either be wave energy period, Te, or wave peak period, Tp, depending on the user specification) are listed along row 5 and must be in seconds. If the locations of these values are modified then the VBA code needs to be changed accordingly (see section 7.6). The resolutions of the steps in Hs and wave period need to match the resolutions provided in the 'Weather' spreadsheet.

The tariff used as the sale price of electricity must be defined by the user in pence per kilowatt-hour (p/kWh).

4.6 DAYLIGHT SHEET

The 'Daylight' spreadsheet contains a matrix (or matrices) showing the amount of hours of daylight in each month for a given location. The resolutions of the hour values must match the time step resolution of the O&M model (defined by the resolution of the weather input data). In the matrix, hours of daylight must read 'Day', with hours of darkness reading 'Night'. These words are used by the VBA code in defining weather windows if the 'night operations' entry in the 'Inputs' spreadsheet has seen set to 'no'. More locations can be added, with information about daylight hours freely available on the internet. The VBA code (primarily section 7.6) must be modified if other array locations are to be analysed.

4.7 WEATHER SHEET

The 'Weather' spreadsheet contains the weather data averaged for each time step and placed into 'bins'. The resolution of the bins for significant wave height and wave period must match the values seen in the power matrix (section 4.5). The wave period can be given as either wave energy period (Te) or wave peak period (Tp) so long as it matches the power matrix and operational limits. The wind speed values are also placed into bins as a consequence of the Markov Chain method of simulating the time series. A full report explaining the methodology and validation of this weather simulation model is available (see the 'Weather Simulation Report', WES, 2017a). The time step is identified by the corresponding year, month, day and hour. The information is printed as indicated in Table 4.2. If this layout changes then the VBA code needs to be modified (section 7.6).

Table 4.2. Layout of the 'Weather' spreadsheet

Column	A	B	C	D	E	F	G
Parameter	Year	Month	Day	Hour	Hs (m)	Period (s)	U (kts)

The VBA code prints out the average power, energy and revenue (array at 100% capacity) at each time step alongside the weather dataset. This allows the user to check that the power matrix is being used correctly. This calculated information is cleared at the beginning of a new simulation.

The 'Weather' spreadsheet will update automatically depending on the user's selections in the relevant entries on the 'Inputs' spreadsheet. If a new dataset is required then the dataset store (an Excel workbook with a different dataset on each spreadsheet) corresponding to the specified array lifetime must also be open. In this case, the VBA code will clear the information contained in the present 'Weather' spreadsheet and will replace it with the new dataset. If the user opts to use the weather data already placed in the spreadsheet but the array lifetime does not match, then an error message will appear explaining the problem.

The user does not need to add any information to the 'Weather' spreadsheet manually. However, if a new site is analysed then the VBA code needs to be modified so that the correct dataset store of weather conditions is identified. The new site needs to be stated in the 'Inputs' spreadsheet. The site also needs to match the 'Daylight' spreadsheet if an analysis of marine operations taking place only in daylight hours is to be undertaken.

4.8 HINDCAST SHEET

The 'Hindcast' spreadsheet contains the original hindcast dataset of weather conditions used to generate the datasets (placed in the 'Weather' spreadsheet) for the assessed array location. The site needs to match up with the 'Weather' and 'Daylight' spreadsheets, as well as the 'array location' entry in the universal inputs sections of the 'Inputs' spreadsheet. The column layout needs to match that seen in the 'Weather' spreadsheet (Table 4.2) and the wave period needs to be in the corresponding format (Te or Tp).

The hindcast dataset is only used by the cost-benefit analysis part of the model to calculate estimated wait times ahead of WEC installation following an offsite repair, and the estimate power at each time step.

5 PROCESSES

The 'Inputs' spreadsheet contains macro buttons for three processes that run the O&M model simulations:

- Fast run
- Full run
- Stat run

5.1 FAST RUN

The 'fast run' uses the information contained within the input spreadsheets to produce the 'Results' spreadsheet containing output data of the simulated array for the specified lifetime. This is the only output of the 'fast run'. The speed of this process varies depending on the selected inputs. One example is that if a 10 WEC array is chosen for a lifetime of 20 years, then the 'fast run' will be completed in approximately 30 seconds. However, if either or both of the cost-benefit analysis options are enabled, then this run time increases significantly.

5.2 FULL RUN

The 'full run' simulates the array lifetime in the same way that 'full run' does. In addition to producing the 'Results' spreadsheet, however, the 'full run' also creates a spreadsheet for every year of the array lifetime showing a detailed breakdown of the O&M logistics occurring at each time step. The 'full run' is useful for VBA debugging as well as providing a visual breakdown of the operations of the wave energy array. The 'full run' process does take significantly longer to complete its simulations than the 'fast run'.

5.3 STAT RUN

The 'stat run' process carries out the same function as the 'fast run' but does so multiple times and produces a series of output spreadsheets containing statistical information, such as mean, maximum and minimum values. 'Stat run' should be the primary process when using the O&M model for analysis. The results of one simulated array lifetime contain a degree of variability due to the Monte Carlo nature of the model. Therefore, it is recommended that at least 50 lifetimes (a.k.a. 'loops') are simulated in order to obtain reliable outputs, as demonstrated by Gray (2017).

A 'stat run' will take approximately the same length of time as each of the 'fast run' simulations for the given inputs, summed for each loop. For example, 50 loops of the 'fast run' described in section 5.1 (each taking ~30 seconds) will take approximately 25 minutes. Upon being pressed, the 'stat run' process presents the user with a message box asking how many loops are required, as shown in Figure 5.1.

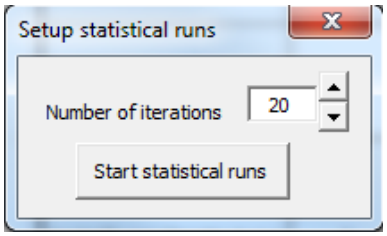

Figure 5.1. Initial message box for the 'stat run' process, requesting the number of iterations required

It is not necessary to complete all loops at once, as the ‘stat run’ offers the user the chance to continue from the loops undertaken so far. The user can choose to do this by selecting ‘no’ when asked if they want to start a new results sheet, as shown in Figure 5.2. Clearly, it is important that none of the inputs are changed between sections; otherwise the results will be invalid.

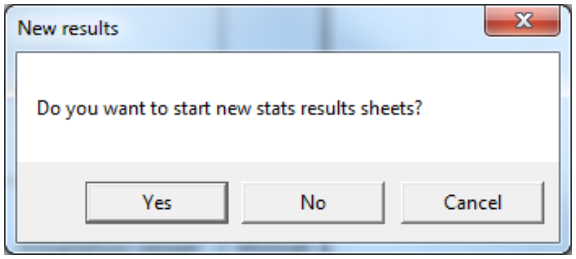

Figure 5.2. Secondary message box for the 'stat run' process, asking if new statistical sheets are required

The ‘stat run’ process recognises the ‘use current weather’ universal input on the ‘Inputs’ spreadsheet (section 4.1.1). If this entry is set to ‘yes’ then the weather dataset stored in the ‘Weather’ spreadsheet is used for all the loops. If it is set to ‘no’ then the dataset store must be open to allow the model to choose a dataset at random at the start of every loop. The ‘choose specific dataset’ universal input must be set to ‘no’ for statistical runs. Therefore, if the current dataset is to be used then it must correspond to the array lifetime.

6 MODEL OUTPUTS

As stated in the previous section, there are three types of output spreadsheets created by the O&M model processes:

- Results
- Run sheets
- Statistical sheets

Output spreadsheets are placed to the right of the 'Inputs' page in the model's Excel workbook. At the start of a new process, all redundant output spreadsheets are deleted in order to avoid confusion between different sets of results. Monetary values are printed in the 'output format' specified in the 'Inputs' spreadsheet.

6.1 RESULTS SHEET

The 'Results' spreadsheet contains the key output information from a model simulation and is created by each of the processes detailed in section 5. The information is presented as a full breakdown showing average values for each WEC in every year of the array lifetime for the following parameters:

- Availability
- Parts costs
- Other costs
- Inspection costs

For each parameter, the WEC values are listed first, followed by an average of all the WECs. The final row is the average value for the entire array. If there are no array-based fault categories or maintenance tasks then the 'All wecs' values will match the 'Array' results. The average results for the years of the array lifetime are printed in the columns, with an annual average printed in the extreme right column.

Below the full breakdown of the four parameters is a table containing revenue information. For each year, as well as an annual average, the sum of the revenue for the following three cases is listed:

- Total revenue earned by the array
- Total revenue that could have been earned by the array if at 100% capacity
- Sum of the lost revenue of the array

The next breakdown shows the fees incurred by hiring external contractors for every year and the annual average. These values will be zero if 'enable short term contractors' is set to 'no' in the 'Labour' spreadsheet.

The 'Vessels' results show each of the boats listed in the order they appear in the 'Vessels' spreadsheet. For each year and for the annual average, three outputs are provided:

- Total hire fees incurred
- Total fuel cost incurred
- Total number of intervals (i.e. time steps) the vessel is in use

Below the vessel details is a table showing the causes of any delays to work. This includes delays to marine operations, as well as delays to undertaking onsite or offsite repairs. The first entry gives the number of instances (note: this is not the number of intervals) where work has been delayed, for each year and the annual average. Below that is a percentage breakdown of the following causes of delays:

- Not enough space at the onshore/quayside O&M base - *space delays*
- No appropriate vessel available - *vessels delays*
- Not enough spare parts are at the O&M base - *parts delays*
- The weather window is closed - *weather delays*
- Lack of available technicians - *technicians delays*

The key information is then brought together in the summary table. For each year and the annual average, the following information is provided:

- Average availability of the array
- Sum of the revenue earned by the array
- Sum of labour cost for permanently employed technicians
- Sum of additional labour cost for external contractors
- Sum of parts costs
- Sum of other costs
- Sum of inspection costs
- Sum of total vessel hire fees
- Sum of total vessel fuel costs
- Total operational expenditure (OPEX)
- Profit (revenue minus OPEX)

The next table contains the following information about all the fault categories:

- Failure ID
- Total occurrence
- Total occurrence repaired
- Parts costs per year
- Other costs per year
- Vessel hire fees per year
- Vessel fuel costs per year
- Total direct costs per year

When repairs are carried out, the vessel hire and fuel costs are assigned to those failures that have occurred. If the repairs had to be undertaken at the O&M base, then the share of these costs is assigned based on the number of days each repair has incurred. If onsite repairs are undertaken then the share is based on the number of hours required to undertake each repair. The total direct costs for each fault category are calculated as the sum of the annual parts costs, other costs, vessel hire fees and vessel fuel costs.

Also within the fault categories output table is information about the causes of lost revenue for each failure. Lost revenue is assigned based on the power loss from each simulated failure whenever overall power loss is reduced. If one or more failures have occurred causing a WEC to be offsite (i.e. producing no power), then the share of lost revenue is assigned accordingly. The lost revenue information is intended to assist decision making about WEC design and O&M strategy;

therefore, the WEC-based fault categories have lost revenue assigned ahead of array-based failures. The total lost revenue per year is provided in the column to the right of the total direct costs. This is broken down further into:

- Annual lost revenue whilst undergoing onsite repair
- Annual lost revenue whilst in transit
- Annual lost revenue whilst offsite
- Annual lost revenue whilst onsite

The sum of the annual lost revenues whilst either offsite or onsite is then broken down further into the causes of the lost revenue:

- Annual lost revenue whilst waiting for space at the O&M base
- Annual lost revenue whilst waiting for vessels
- Annual lost revenue whilst waiting for spare parts
- Annual lost revenue whilst waiting for a weather window
- Annual lost revenue whilst waiting for technicians
- Annual lost revenue whilst the WEC is onsite but not set for repair
- Annual lost revenue whilst undergoing offsite repair

At the end of the 'fast run' and 'full run' processes, the table of fault categories is rearranged in order of the total direct costs per year, with the largest first. For statistical runs, however, the table is kept in the order they are listed in the 'Inputs' spreadsheet.

The final table in the 'Results' spreadsheet details the same information as the fault categories but for the scheduled maintenance events. The only differences are that there is only one 'occurrence' value and 'inspection costs per year' are included.

Three graphs in histogram form are created in the 'Results' spreadsheet following either a 'fast run' or a 'full run'. The first shows the share of annual OPEX incurred by each fault category, not including labour. This information is taken from the 'total direct costs' column in the fault category output table. The second graph shows the share of annual lost revenue for each fault category, using the 'lost revenue total' column. The final graph breaks down the total annual amount of lost revenue (i.e. from all fault categories) into the causes, including whilst being repaired and in transit. The 'cost-benefit delay' data is taken from the 'total' value in the 'lost rev onsite not set' column. The graphs are located in the top-left corner of the 'Results' spreadsheet.

6.2 RUN SHEETS

'Run sheets' are only created by the 'full run' process. They provide visual representation of the lifetime operations and maintenance of the array and can assist in code debugging and validation. The sheets are named 'Year 1', Year 2' etc. up to the number of years specifying the array lifetime.

The first four columns provide information about the date at each time step. The 'interval' is a continuing value throughout each year. The 'month', 'day' and 'hour' values follow the same convention as seen in the 'Weather' spreadsheet (see section 4.7). The months start in December for every year in order to group meteorological seasons together if required (as discussed in section 4.1.3). The 'run sheets' utilise Excel's 'freeze panes' functions so that the date values and the headers are always on show, no matter where the user scrolls on the page.

The two columns to the right of the 'date' section provide information about the state of the array. This includes showing the occurrence of array failures, as well as the power capacity of the array at each interval. The 'array failures' cells will say 'operating' unless there are any simulated array-based failures. These will be listed when they occur. The 'array failures' cells will also indicate when array-based repairs or maintenance tasks are being undertaken.

To the right of the 'array' section is detailed information about the state of each WEC at every interval. The 'failures' column is used to indicate whether the WEC has suffered any failures and will show when, and how, the WEC is being repaired. If the WEC has not suffered any failures then the cell will read 'fail:' with no colour fill. However, if the WEC has suffered one or more failures, then the cell will show a list of failure IDs and will be filled with the same colour as the classification of the most severe failure (i.e. red for major, amber for intermediate, green for minor). Intervals where a marine operation is being undertaken will be filled grey and will say either 'being removed' or 'being installed'. When a WEC is offsite, the cell will say 'off site' and be coloured dark blue. If a WEC has failures that are being repaired offshore (i.e. onsite) then the 'failures' cell will read 'under repair' and will be coloured dark red. The adjoining column, labelled 'maintenance', contains information about scheduled maintenance events. If no WEC-based maintenance events are due to take place then the cell will read 'not due' with no colour fill. However, if any event is due then the cell will be coloured red and read 'due:' followed by the IDs of any scheduled maintenance categories that are due. For scenarios when the 'no WECs allowed at base for maintenance' entry on the 'Inputs' spreadsheet (see section 4.1.1) limits scheduled maintenance, then the cell will read 'delay retrieval for maint:' with a list of the categories and will be coloured amber.

The next section shows output information for all the vessels listed in the 'Vessels' spreadsheet. For each vessel, the name is shown above three columns labelled 'state', 'hire fees' and 'fuel cost'. At each time step, the 'state' cell is either coloured green and read 'not in use', or coloured red and say 'in use'. The 'hire fees' and 'fuel cost' cells keep track of the cumulative value of their respective parameters and show the values in pounds sterling (i.e. the working monetary format of the model).

Columns relating to the 'weather windows' are located next to the 'vessels' section. There is a column for each of the operational limits types defined in the 'Ops Limits' spreadsheet (section 4.4). The header is labelled 'Ops 1', 'Ops 2', etc. At each time step, the cell reads 'CLOSED' and is filled red if the weather conditions exceed the defined limits. Alternatively, the cell is filled green if the weather window is 'OPEN'.

A column is provided for each technician permanently employed at the O&M base, as defined in the 'Labour' spreadsheet (section 4.3), next to the 'weather windows' section. At each interval, the cell is filled red and reads 'busy' if the technician has been assigned to a task. Otherwise, the cell is green and says 'free'. The final column in the 'Technicians' section contains basic information about external contractors. If no contractors are being used then the cell reads 'no' and is filled green, otherwise, the cell is red and reads 'yes'. There is no output for the number of contractors used at each time step, although this functionality could be added if deemed useful.

The two columns on the far right of the 'run sheets' contain the number of spare parts located at the O&M base at each interval. The two options as defined in the 'Inputs' spreadsheet (section 4.1.1) are 'PTO unit' and 'instrumentation box', although this can be expanded if relevant.

6.3 STATISTICAL SHEETS

The ‘stat run’ process produces the following output spreadsheets, in addition to the ‘Results’ spreadsheet:

- Stat_mean
- Stat_max
- Stat_min
- Stat_range
- Stat_results

At the start of a ‘stat run’, the user is asked for the number of ‘loops’ they wish to simulate, as stated in section 5.3. A ‘fast run’ process is undertaken for each of the requested loops, producing a ‘Results’ spreadsheet, as detailed in section 6.1. The table of fault categories is kept in the same order as listed in the ‘Inputs’ spreadsheet. At the end of the first loop, the ‘Results’ values are copied into each of the ‘stat_mean’, ‘stat_max’, ‘stat_min’ and ‘stat_range’ spreadsheets. For each subsequent loop, the statistical parameters inferred by the sheet names are calculated. Therefore, these four output spreadsheets are presented in exactly the same format as ‘Results’ but provide statistical information about the simulations. The three graphs that are printed to the ‘Results’ sheet for the ‘fast run’ and ‘full run’ processes are instead printed to the ‘stat_mean’ spreadsheet.

The values for average availability, revenue and OPEX in each year of the array lifetime produced by each simulated loop are printed to the ‘stat_results’ spreadsheet. These values are also averaged across the lifetime of the array for each loop, and shown alongside the annual average profit on the far right of the spreadsheet, as seen in Figure 6.1. The name of the weather dataset used for each simulated lifetime is stored in column B. The mean values from all the loops are calculated for the four parameters and can be checked against the ‘stat_mean’ sheet. The 95% confidence intervals are calculated using the following equation:

$$95\%\ confidence\ bounds = \bar{X} \pm \frac{z \times \sigma}{\sqrt{n}}$$

Where $\bar{X}$ = mean, z-value = 1.96 (for 95% confidence), σ = standard deviation, n = population size

Loop	Dataset	Year 1			Year n			Average			
		Avail	Rev	OPEX	Avail	Rev	OPEX	Avail	Rev	OPEX	Profit

Figure 6.1. Layout of 'stat_results' output spreadsheet, where n = array lifetime

The information presented in the ‘stat_results’ spreadsheet is used to create six charts which provided a visual representation of the results of the statistical simulations. The four charts printed at the far left of the spreadsheet show the annual results of each simulated lifetime in terms of availability, revenue, OPEX and profit across the lifetime of the array. The dominant visual in each chart is the bold line showing the average annual values of all loops. To the right of these are two more charts. The top one shows the cumulative profit earned throughout the array lifetime for each loop, again with the annual values represented by the bold lines. The histogram below the cumulative profit chart shows the annual average monetary values of revenue, OPEX and profit, with the 95% confidence bounds applied.

7 MODEL ALGORITHMS AND VBA CODE

As described previously and shown in Figure 3.1, the inputs to the model are stored in spreadsheets in a Microsoft Excel workbook. The output information is also printed to spreadsheets. However, the actual functionality of the O&M model is carried out in Visual Basic for Applications (VBA); the programming language supporting Excel. The VBA part of the model has been creating using an Object Oriented Programming (OOP) structure. This is necessary in order for the same functions to be utilised for every time step throughout each year of the array lifetime. It also allows the same series of functions to be used for each WEC and vessel in the array. This structure can be seen in Figure 7.1 showing most of the 'modules' and 'class modules' contained within the model. Modules are 'objects' containing any number of 'procedures' which are used to undertake specific actions when 'called'. Procedures can either be 'functions' or 'subroutines'. These are essentially the same thing, the only difference being that functions can 'return' a value to the calling procedure. A class module is also an object; however, it is lower down in the VBA hierarchy than a module and can therefore be created multiple times if required (i.e. for each WEC is the array). 'Variables' are used throughout the VBA code to store information in various types. Data types could include 'integer' (whole number between -32,768 and 32,767), 'string' (text), 'long' (whole number up to 2 billion) or 'double' (a decimal number).

This chapter details every aspect of the VBA code by going through each object in turn and describing every procedure. Flowcharts are used throughout the chapter to show how the procedures fit together to form the O&M model. The notation follows the same structure as the VBA language, where the object name is followed by a full stop and the name of the procedure (i.e. *object.procedure*). The names of objects, procedures and variables are printed in italic script. The objects are discussed in order of their hierarchy in the model code, where possible. This is intended to make the structure and functionality as clear as possible, thus providing a means of enabling future modifications as well as model validation. Data types are printed with an uppercase first letter (e.g. Integer, Double etc.)

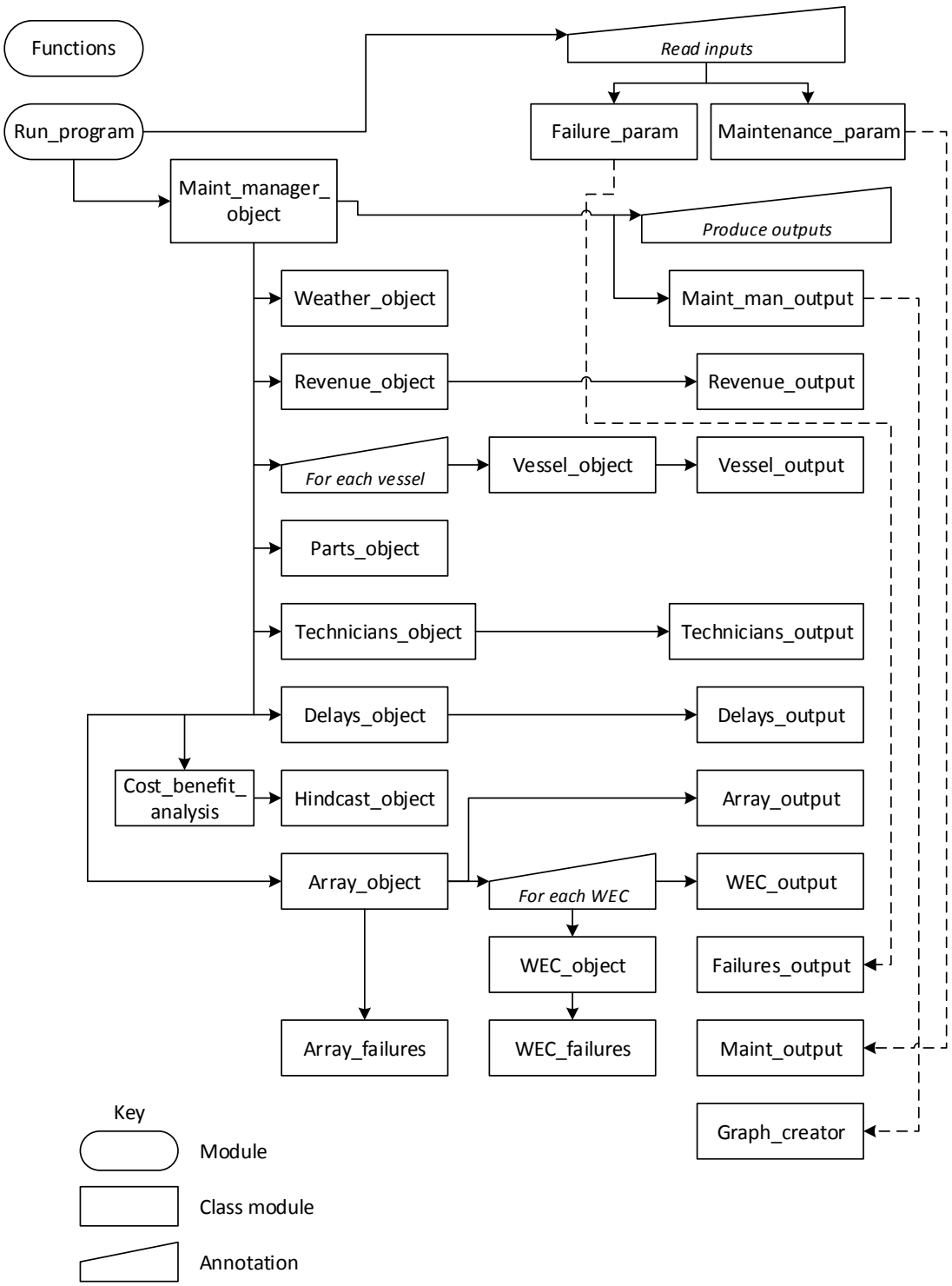

Figure 7.1. Object Oriented Programming structure of the VBA-based O&M model, showing key modules and class modules

7.1 FUNCTIONS

The object *functions* is a module containing key procedures that are used by a number of other objects at various times of the model simulations. Whilst VBA has a large amount of in-built functions, these do not cover everything that the O&M model needs to use. *Functions* also contains new data types that are used in the model for defining variables.

7.1.1 Defining new types

It is often useful to create new data types in addition to those stored in VBA (e.g. Integer, Double etc.). For each of the five new data types defined, the term *Enum* is used. This means that the entries in the data types can be referred to either by their name or by their value. The value is determined by the order they appear in the *Enum* section, starting at zero.

The *severity* defines the classification of faults and can either be *major, intermediate* or *minor*.

The *vessel_state* asserts that a vessel can either be *not_in_use* or *in_use.*

The *array_state* at any time can either be *operating* or *being_repaired*.

In contrast, the *WEC_state* can be one of five options at any time. If the WEC is at the onshore or quayside O&M base then it is *off_site*. If the WEC is in transit then it is either *being_removed* or *being_installed*. If offshore work is being undertaken (i.e. a parts replacement) then the WEC is *under_repair*. Otherwise, the WEC is *on_site*.

The *yes_no* data type can either be *yes* or *no*. Although this is binary in a similar way to Boolean (i.e. True or False), it has been included for situations where a *yes* or *no* answer is more readable to the user than *True* or *False*.

7.1.2 Insert sheet

The *insert_sheet* subroutine is used to check if a particular worksheet (a.k.a. spreadsheet) exists in the workbook. If it doesn't exist then it is inserted and positioned at the end (far right) or the workbook.

To achieve this, *insert_sheet* takes the arguments of *book* and *sheet* from the calling procedure. These correspond to the names of the workbook and the worksheet. The variable *exists* is defined as Boolean and *i* is used as the Integer counter. *Exists* is first initialise to False, stating that the worksheet does not exist in the workbook. The VBA term *with* is used to avoid unnecessary multiple instances of the higher level in-built functions *workbooks* (a collection of open workbook) and *sheets* (a collection of worksheets within that workbook). A *for* loop is used to check each worksheet by using the *sheets* function *count* to identify the number of sheets in the workbook. The *sheets* functions *item* and *name* are then used to identify the sheet that matches the requested worksheet name (*sheet*). When found, *exists* is changed to True and the *for* loop is exited to save time. After the loop, if *exists* is still False then the sheet is added (using the *sheets* function *add*), renamed, and then moved to the end of the workbook.

7.1.3 Timer

The *timer* function is used to calculate the length from a defined *start_time* (sent from the calling procedure) until the current time. It converts the calculated time into a readable format.

The *end_time* variable has no defined data type as it is set to be the current time using the in-built function *Now*. The *time_in_secs* is a Long variable and is calculated using the in-built function *DateDiff*. This requires the two times, *start_time* and *end_time*, as an input along with the required output format 's' (meaning seconds). The hour, minute and second values are all defined as Integer and named *hr_val, mnt_val* and *sec_val* respectively. An *if* condition is then used to determine which in format to return the time. If *time_in_secs* is less than 60 then only seconds are necessary. On the other hand, if *time_in_secs* is greater or equal to 3600 then the format needs to be in hours, minutes and seconds. Lastly, only minutes and seconds are needed if the *time_in_secs* is between 60 and 3600. Throughout these calculations, the in-built function *Int* is used to return the integer part of a number. The procedure is a function because it returns a String value back to the calling function by setting its own name, *timer*, to be the formatted time values.

7.1.4 Delete run sheets

The *delete_run_sh* subroutine loops through all the worksheets in the workbook and deletes the 'Results' and 'run_sheet' spreadsheets described in section 6.

The data type *worksheet* is used to loop through each sheet in the collection *workbook.worksheets*. The function *DisplayAlerts* is set to False in order to avoid Excel printing an error message when trying to delete a worksheet containing information. Throughout the loop, if the worksheet starts with either 'year' or 'Results' then it is deleted. The in-built function *Left* is used to identify a section of a String with a specified number of letters. The *sheets* function *Delete* is used to delete the worksheet. At the end of the subroutine, *DisplayAlerts* is reset to True to allow the model to show error messages again.

7.1.5 Delete statistical sheets

The *delete_stat_shts* subroutine follows exactly the same structure at *delete_run_sht* (section 7.1.4). However, instead of searching for worksheets that begin 'year' or 'Results', it deletes all sheets that start with 'stat', thereby deleting all the statistical sheets described in section 6.3.

7.1.6 Delete this sheet

The *delete_this_sht* subroutine also loops through each worksheet in the workbook in the same way as described in sections 7.1.4 and 7.1.5. However, rather than search for a sheet whose name starts with certain words, it searches for the name of a worksheet that matches the argument *this_sht*. Again, *DisplayAlerts* is used to control and reset Excel's ability to show error messages.

7.1.7 Max and min

Excel worksheets have in-built functions to calculate the maximum and minimum of a range of values, named *MAX* and *MIN* respectively. For VBA to use these functions, however, the code needs to first use the higher level functions *Application* and *WorksheetFunction*. This can make the code unnecessarily long, especially when find maximum and minimum values is quite a common feature of the O&M model. Therefore, the functions *max* and *min* have been created to find the maximum and minimum respectively of two input values, named *var1* and *var2*. The function names are set to be either *var1* or *var2* and returned to the calling function.

7.1.8 Terminate program

The *terminate_program* subroutine is used throughout the VBA code to prompt the user to exit the simulations. The counter *i* is defined as an Integer and used to loop *for* values up to 200 (a nominal figure). At each step, the user is prompted to exit the program by pressing Ctrl + Break. It is useful to note that not all keyboards have a Break button. Instead, the keyboard may have a Pause button to carry out the same functions. In some keyboards, however, neither of these buttons exists. In these situations, a combination of the function key (Fn) + P, or Fn + Alt + P, needs to be used.

7.1.9 Is in array

The *is_in_array* function is mused to determine if a String value is stored in a certain array. It is sent the variables *this_string* and *arr* to achieve this by the calling procedure. The Integer value *i* is used as a counter. The return Boolean value *temp_bool* is initialised to be False. It is generally considered good practice to avoid changing the value of the function itself throughout the procedure. A *for* loop goes through each entry in the array (*arr*) and changes *temp_bool* to True if *this_string* matches the entry. Finally, the return value *is_in_array* is set to *temp_bool*.

7.1.10 String array and 2d array

For debugging purposes it is often quite useful to know exactly what values an array contains. The two functions *str_array* and *str_2d_array* take one dimensional and two dimensional arrays respectively and converts the entries into a readable format for printing in Excel's message boxes. To achieve this, they take the array itself (*this_array*) and the array's variable name (*arr_name)* as arguments from the calling procedure. A temporary return value, *str*, is defined with the String data type and initialised to read the *arr_name,* accompanied by appropriate formatting. Each entry in the array is then considered in a *for* loop using the counter *i*. The *str_2d_array* function also requires an additional counter, *k*, to deal with the extra dimension. Each entry is added to *str* with the appropriate formatting. The return value (i.e. the name of the function) is then set to *str*.

7.1.11 Number of rows

The function *num_rows* is used to obtain the total number of rows in a spreadsheet that contain data. The worksheet *mysheet* and the maximum row to search (*max_row)* are sent by the calling procedure. The Long temporary return value, *no_rows*, is initialised to zero. Each row, from 1 to *max_row*, is considered in a *for* loop using the counter *i*, defined as Long. The *WorksheetFunction CountA* is used to count the number of cells in the row that contain any kind of data. For each row that does contain data, one is added to the value of *no_rows*. When a row is found that contains no data (i.e. *CountA* = 0), then the *for* loop is exited. *num_rows* is set to be *no_rows* and returned to the calling procedure. The limitation of the function is that it will not operate as planned if a worksheet has been formatted to have, for example, one empty row between the headers and the main text for readability purposes. This needs to be carefully considered when adding any line of code to the O&M model which calls the *num_rows* function.

7.1.12 Order a 2d array

The cost-benefit analysis part of the model, previously discussed in section 4.1.1, uses the function *get_ordered_array_2d* to sort a two dimensional in a particular order. The array that needs to be sorted in sent to the function as *OrigArray*. It is a 2D array where the first dimension is the number

of WECs in the array and the second dimension contains information about each WEC. This information is discussed later in section 7.12. To explain *get_ordered_array_2d* at this stage, it is important to know that the information stored in the array for each WEC consists of the device ID, followed by numerical values for each of the remaining entries. These entries refer to a particular aspect of repairs or maintenance and are entered in the required order by which to sort.

The *get_ordered_array_2d* function starts by using variables, *no_mach*, *lower_orig2* and *upper_orig2* to define relevant boundaries of the array. A one dimensional array, *hold_array*, is created with the boundaries *lower_orig2* and *upper_orig2* and is used to store entries from a single WEC on a temporary basis. The return value, *temp*, is initialised to be the *OrigArray*. The function then only proceeds if there is more than one WEC in the array (*if no_machs* > 1).

Note: 'array' in this context refers to a virtual array created by VBA, not the wave energy array. This is an important distinction and will be made clear throughout this document.

Each entry of the 2D array is considered in a *for* loop, starting at *upper_orig2* and moving backwards through the array using the *Step* -1 feature (ignoring the WEC ID entry) with the counter *this_column*. Looping backwards is this manner is a useful approach for when entries in an array or in a spreadsheet are repositioned or deleted. In this context, however, it means that the last entry of the 2D array (i.e. the one with the lowest priority) is ordered first. This method leaves the highest priority entry to last, producing a fully ordered array. For each *this_column*, a *Do While* loop is utilised to undertake the subsequent actions until all required swapping of entries for that column has been completed. The Boolean variable *still_swaps_needed* is first initialised to True. The Integer counter *i* is then used to loop backwards through each WEC in the array. If the next WEC in the array has a small value (in *this_column*) than the one under consideration, then *still_swaps_needed* is set to False and the values of *this_column* are moved up the array. The repositioning of the array is achieved by using *hold_array* to store a line of information and, one at a time, replace the previous line of values in the *temp* array. When this happens, the *for* loop moves onto the next *this_column* entry of the array. The returned 2D array, *get_ordered_array_2d*, is set to be *temp.*

7.1.13 Workbook open

The Boolean function *WorkbookOpen* is used to identify whether or not a workbook with a certain name (*FileName)* is open. *WorkbookOpen* is first initialised to True, stating that the requested workbook is open. If the function tries to *Activate* a workbook that is not open then an error message will be displayed. This knowledge is used so that when an error would otherwise be displayed, the feature *OnError GoTo* sends the code to the *NotOpen* section of the function. In *NotOpen*, the return value (*WorkbookOpen*) is set to False and the calling function is told to *Resume Next* actions. If the workbook is open, however, then it is activated successfully and the error handling feature is reset using *On Error GoTo 0*.

7.1.14 Column letter

The String function *Col_Letter* is used to convert the ID of a column into the letter reference used by Microsoft Excel. This is useful due to that fact that VBA refers to columns by a number (i.e. 1, 2, 3 etc.), rather that the Excel letter system (e.g. A, B, C etc.). To achieve this, *Col_Letter* takes the number reference of the column (*IngCol*) and uses the in-built VBA function *Split*. The String value used in *String* is the *Address* of the first cell is the required column. This uses the *Cells* reference

system, whereby the first entry is the row reference and the second is the column number reference. The cell *Address* is obtained with Excel's 'absolute' notation ($), which is used by *Split* to find the column letter.

Note: *Cells* is a function used extensively throughout the O&M model VBA code. To refer to a cell from an Excel spreadsheet, the syntax is *Cells(row reference number, column reference number)*. This may be counter-intuitive to Excel users who are used to referring to cells by their column reference letter first, followed by the row reference number (e.g. A1, B5 etc.).

7.1.15 Delete charts

The function *delete_charts* searches through a specified worksheet (*this_sheet*) and deletes all charts that exist in the sheet. The data type ChartObject is used with its function *Delete*. A *for each* loop can be used to search through the spreadsheet because ChartObjects is a high-level collection (in a similar way to Workbooks and Worksheets). Although this procedure is a function, it does not return a value. In this regard, it could just as readily be defined as a subroutine.

7.1.16 Find index reference

The function *find_index_ref* is used to find the either the row or column reference of a cell containing a certain value within a worksheet. The function is sent information pertaining to the item ('row' or 'column') the calling procedure wants to find (*find_this*), the ID reference of the known row or column (*index_const*), the value to find (*search_text*), and the name of the worksheet (*sht_name*). The temporary return value, *ret_val*, is initialised to zero. An *if* condition is used to distinguish between the two options of *find_this*. In either case, an Integer counter, *i*, is used to loop through each cell in the known row or column (*index_const*). The in-built functions *Cells* and *Value* are used to identify the cell where *search_text* is found. The temporary return value, *ret_val*, is then assigned to the reference of the unknown row or column, and subsequently defines the final return value, *find_index_ref*.

7.1.17 Round all decimals

The function *round_all_decimals* is used to present to results of the O&M model in a readable manner. The calling procedure sends the function the name of the sheet it wants to format (*sht_name*). The Long variables, *x* and *y*, are used to refer to the numerical reference of columns and rows respectively. The variable *cell_val* is defined as a Variant data type because it is assigned to each cell in the sheet in turn, the format of which may vary. The maximum numbers of rows and columns to search through are defined using the term *Const* and set to be the values *max_x* and *max_y* respectively. The number of required decimal places to format is also defined as a *Const* using the variable name *num_dps*. The sheet name is activated and each cell is selected in turn. If the cell *IsNumeric*, is not *IsEmpty* (in-built VBA functions), and is not an integer, then the correct format is applied using the *Cells* function *NumberFormat*. Only five decimal places are currently accounted for, but this can be added to if required.

7.2 RUN PROGRAM

The primary module of the O&M model VBA code is named *run_program*. It defines a number of *Global* parameters (variables that can be used any class module), sets up the model, and controls the simulation processes.

7.2.1 Global variables and constants

At the top of the *run_program* module there are a series of variables defined as particular data types or as constant values. The majority of these variables are defined using the term *Global*, meaning that other modules and class modules can use the information stored by using the variable names. A small number of variables here are not *Global*, indicating that they are only used in the *run_program* module itself.

The names of the primary input spreadsheets are stored in variables:

- *workbook_name* - stores the name of the active workbook
- *data_sheet* - stores the name of the 'Inputs' spreadsheet
- *vessels_sheet = "Vessels"* - name of the 'Vessels' spreadsheet
- *labour_sheet = "Labour"* - name of the 'Labour' spreadsheet
- *ops_limits_sheet = "Ops Limits"* - name of the 'Ops Limits' spreadsheet
- *power_sheet = "Power"* - name of the 'Power' spreadsheet
- *weather_sheet = "Weather"* - name of the 'Weather' spreadsheet

Useful variables and constants defined are:

- *data_col* - the column ID reference of the universal inputs in the 'Inputs' sheet
- *time_step* - the resolution of the model in hours (must match the weather data)
- *no_intervals* - stores the number of intervals in a year
- *no_run* - stores the array lifetime in years
- *run_sheet* - the prefix text of the output 'run sheets'
- *speed* - determines whether to undertake a 'fast run' or 'full run' process
- *normal_run* - distinguishes between statistical runs and other processes
- *num_vessels* - stores the number of vessels in the 'Vessels' spreadsheet
- *num_technicians* - stores the number of technicians in the 'Labour' spreadsheet
- *short_term_contractors enabled* – reads the contractor selection from 'Labour'
- *max_wecs_offsite* – store the user selection on space at the O&M base
- *max_wecs_offsite_maint* – stores the user selection on space at the O&M base just for scheduled maintenance
- *CBA_retrieval* - stores the user selection on cost-benefit analysis for WEC retrieval
- *CBA_onsite* - stores the user selection on cost-benefit analysis for onsite WEC repairs
- *CBA_allowance_days* - stores the user selection on delaying CBA decisions if the WEC is due scheduled maintenance within a certain number of days
- *night_ops_on* - stores the user selection on daylight marine operations constraints
- *this_location* - stores the user selection of the array location

Two variables are used to assist the weather selection aspect of the model:

- *store_name* - stores the name of the weather dataset used for the simulation
- *choose_specific* - stores the user selection on choosing a specific weather dataset

Several class modules are defined using the *As New [object]* function:

- *maint_manager* - a new *maint_manager_object* class module
- *fail_param_list* - a new *failure_param_*list class module
- *maint_param_list* - a new *maintenance_param_list* class module
- *fail_output_list* - a new *failure_output_list* class module
- *maint_output_list* - a new *maint_output_list* class module

Finally, some variables are used for storing output information:

- *results_sheet = "Results"* - name of the 'Results' spreadsheet
- *output_money_format* - stores the user selection on format of monetary outputs
- *output_money_divider* - used to convert pounds into the required monetary format
- *graph_creator* - a new *graph_creator* class module
- *start_time* - stores the time at which a simulation began

7.2.2 Start lifetime simulation

The process 'fast_run' and 'full_run', previously described in section 5, are assigned to the subroutines *fast_run_sub* and *full_run_sub* respectively. In each case, the subroutine *run_multi* called and is sent a value pertaining to the required *speed* of the process – 1 for a 'fast run' and 0 for a 'full run'.

At the start of the *run_multi* subroutine, the *start_time* variable is set to be the current time using the in-built function *Now*. The *ScreenUpdating* function is set to False to stop Excel showing the model operations in an effort to cut down simulation time. As this subroutine is only called when the process buttons on the 'Inputs' spreadsheet are pressed, it is valid to set *workbook_name* and *data_sheet* to the names of the *ActiveWorkbook* and *ActiveSheet* respectively. The array lifetime is read from the *data_sheet* and stored in the variable *no_run_loc*. The *Global* variable *speed* is set to be the argument *speed_loc*, sent by *fast_run_sub* or *full_run_sub*, so that it can be used by other objects. *normal_run* is set to True; the purpose of this is made clear in the next section. To avoid confusion between the outputs from this process and any previous runs, all statistical sheets are deleted by calling *functions.delete_stat_shts*. The main program, *run_om*, is then called with the number of years (*no_run_loc*) as an argument. Following the model simulation, the output table of fault categories in the 'Results' spreadsheet is sorted in order of the greatest total direct costs incurred per year, as described in section 6.1. This is achieved by calling the function *sort_fails_table* located in the *fail_output_list* object (section 7.25.10), and uses *functions.find_ index_ref* to find the row reference of the table headers. The *master* control procedure in the *graph_creator* class module is then called to produce the graphs described in section 6.1. Finally, a message box prints the total time that the simulation has taken in a readable format using *functions.timer* with *start_time*. The in-built function *MsgBox* is a common feature of the VBA code used to display information to user. This is particularly useful in the contexts of error handling and debugging.

7.2.3 Run main program

The *run_om* function can be considered the primary procedure in the model's VBA code. It is used to set up the model and carry out the VBA procedures for every time step in each year of the array lifetime, before printing the results.

The identifiers *irun* and *this_interval* are first defined as the Integer data type. If the *normal_run* variable has been defined as True by the calling procedure (e.g. with the 'fast run' and 'full run' processes) then a progress bar is shown at the bottom left of the Excel workbook using the function *DisplayStatusBar*. This *StatusBar* is set to read "Initialising..." at first. The term *Randomize* needs to be used so that a different seed value is produced whenever a random number is set up by the code. This is vital to ensure that failures are simulated differently in every process, thereby making each new model run unique. The *no_run Global* variable, defining the array lifetime, is set to the value sent by the calling procedure. The *copy_weather_data* function is call in order to use a new dataset of weather conditions for the run if required. This is explained further in section 7.2.6. If *copy_weather_data* returns the value zero, then an error has occurred and *functions.terminate_program* is called to prompt the user to exit the program. Every existing output sheet is removed to avoid confusion by calling *functions.delete_run_sh*. Before the main loop of the function, the procedure *setup_class* is called (with the number of *years* in the array lifetime as an argument) in order to set up and initialise each relevant class module. This is described in the next section.

The main function of *run_om* occurs in a nested *for* loop (i.e. a loop within a loop). Each year in the array lifetime is considered using the identifier *irun* from 1 up to the specified number of *years*. The *StatusBar* located at the bottom left of the Excel workbook is updated for the current *irun* if the calling procedure has set *normal_run* to True. The *DoEvents* in-built function is used to ensure the progress bar is updated correctly. If a 'full run' process is taking place (i.e. if *speed = 0*) then the 'run sheets' described in section 6.2 are inserted by calling *maint_manager.insert_sheet_maint_man*. The nested *for* loop then considers every time step in that year (up to *no_*intervals) with the identifier *this_interval*. For each time step, the following procedures are called from the *maint_manager* class module:

- *determine_failure* - carry out Monte Carlo analysis to simulate failures
- *determine_fix* - determine which WECs require maintenance or repair
- *determine_actual_fix* - simulate marine operations and repair/maintenance tasks
- *print_interval* - print to 'run sheets' if 'full run' is taking place
- *next_interval* - set the model up for the next interval

In each case, the called procedure is sent the variables *irun* and *this_integer* in order to recognise the current date. The *StatusBar* is then updated (if required) to read "Printing results", before the outputs are calculated and printed by calling the *post_process* procedure.

7.2.4 Set up class

The subroutine *setup_class* is used to create and initialise all the class modules in the VBA code. Firstly, the total number of intervals in a year (*no_intervals*) is calculated using the defined *time_step*. Values for the number of WECs in the array (*no_total_wecs*), the number of fault categories (*no_fail_loc*) and the number of scheduled maintenance events (*no_maint_loc*) are read from the relevant *Cells* in the *data_sheet*. Four *Global* variables are then stored by reading from the appropriate sheets:

- *num_vessels* - 'Vessels' sheet
- *num_technicians* - 'Labour' sheet
- *max_wecs_offsite* - *data_sheet*
- *max_wecs_offsite_maint* - *data_sheet*

The Boolean variables determining the users choice of whether or not to incorporate a cost-benefit analysis into the model, *CBA_retrieval* and *CBA_onsite*, are initially set to False. If the relevant cells reads 'Yes' then the variables is changed to True. The value of *CBA_allowance_days* is read directly from the *data_sheet*. The method of initialising a Boolean variable before reading the relevant cell is also used to define *night_ops_on*. The difference here being that the variable is first set to True and only changed to False if the relevant cell reads 'No'. These default assignments are made so that the model runs as quickly as possible in the event of an alternative entry. As described in section 4, dropdown lists are used through the input spreadsheets to avoid any invalid entries.

The value for *output_money_format* is read from the relevant cell in the *data_sheet*. Knowing that the monetary inputs to the model are all given in pounds sterling, the *output_val_divider* value is set to the appropriate number (i.e. 1 if '£', 1000 if '£k', 10x10^6 if '£m'). If a value outside the dropdown list is entered then an error message is produced and the user is prompted to exit the program (via *functions.terminate_program*).

Following this process of reading and defining variables based on the input information, *setup_class* then initialises the class modules, as shown in Figure 7.2.

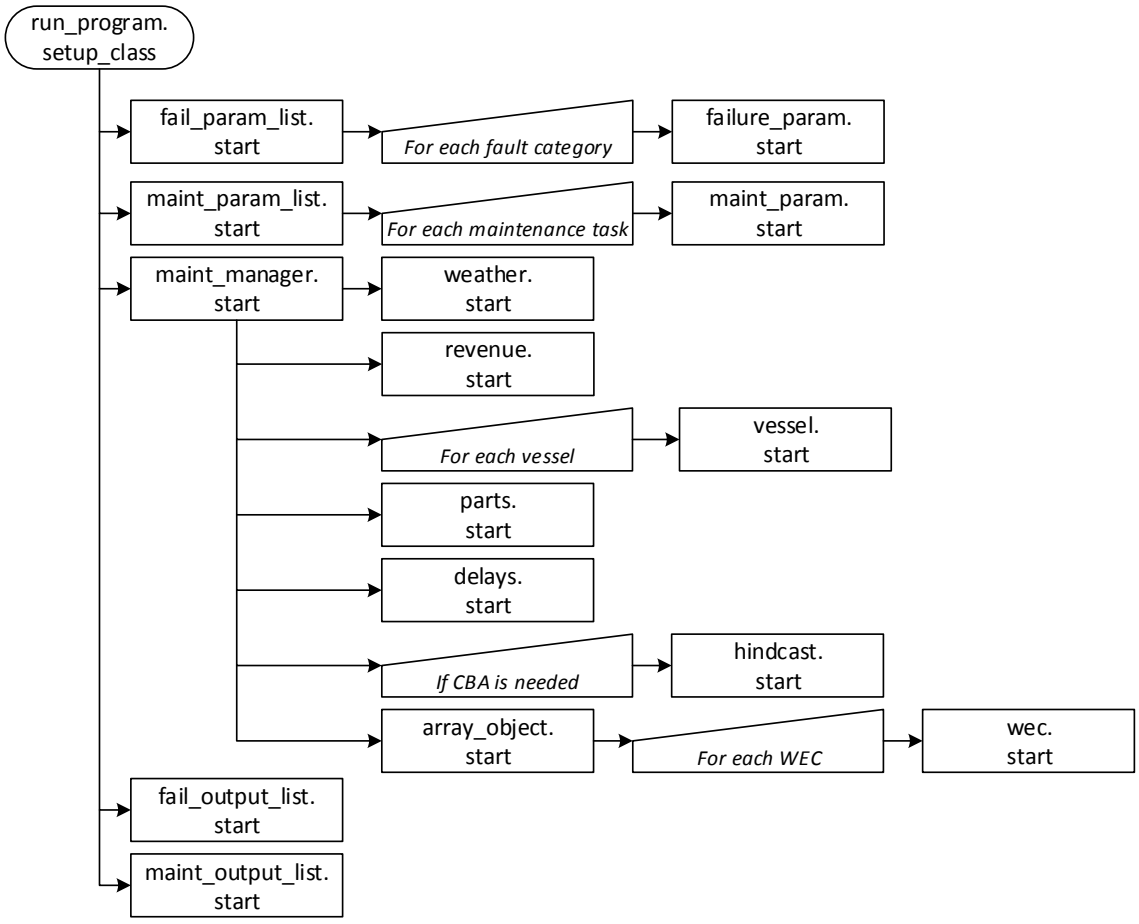

Figure 7.2. Flowchart of the *setup_class* procedure

7.2.5 Post process

Throughout the VBA code, *post_process* procedures are used to produce the outputs described in section 6. The *run_program*, the *post_process* subroutine first uses *insert_sheet* to add the 'Results' spreadsheet to the workbook. The in-built *Cells* function *Clear* is used to wipe all data from the sheet (although none should exist due to the use of *delete_run_sh* in *run_om*). The function *post_process* in the *maint_manager* object is then used to produce the bulk of the results, as well as returning a row reference on which to start printing the next outputs (*start_row)*. The output tables of fault categories and maintenance events are printed using the *draw* procedures in the *fail_output_list* and *maint_output_list* objects respectively. Finally, the sheet is tidied up using the created *function.round_all_decimals* (see section 7.1.17).

7.2.6 Copy weather data

The function *copy_weather_data* determines which dataset of weather conditions to use for the requested simulations and undertakes necessary error handling. It also stores the name of the weather dataset as the variable *store_name*.

The return value, *ret_val*, is initialised to 1 to tell the calling procedure that the function has operated successfully. If an error is found then *ret_val* is set to 0. The next step is to read the array location selected by the user in the 'Inputs' spreadsheet (i.e. *data_sheet*) and store it as the variable *this_location*. *copy_weather_data* then converts this information into an acronym relating to the *site*. The only location entry coded for in the current model is 'North Scotland', which uses the acronym 'FPT' (relating to the WES-owned hindcast dataset for the Farr Point site). If time series' of weather conditions for other sites are obtained then this code needs to be modified appropriately. Relevant error messages and information is displayed if *this_location* is not recognised.

The defined *site* is subsequently used to identify the name of the workbook containing the Markov-generated time series' for the array lifetime (*no_run*). This is stored in the String variable *weather_store_workbook*. The value of *no_run* is also then used to define the first parts of the spreadsheet names (i.e. datasets of weather conditions) stored in the *weather_store_workbook*. This is assigned to the String variable *start_store_name*.

Note: it is important that the user of the Markov Chain Model (see 'Weather Simulation Report', WES, 2017a) understands the required naming conventions defined here when creating the time series'.

The user-defined entries of *use_current_weather* and *choose_specific* are read from the universal inputs column (*data_col*) of the 'Inputs' spreadsheet. If the user has chosen to not use the current weather (i.e. if *use_current_weather = "No"*), then they are prompted to open the *weather_store_workbook* if it not already (using *WorkbookOpen*). When the workbook is open, it is activated and the number of existing sheets is obtained (*worksheet_count*). If the user has chosen to select a specific dataset (i.e. if *choose_specific = "Yes"*) then a message box appears asking the user choose a dataset from a dropdown list. This is achieved by looping *for* each of the datasets with the identifier *this_dataset*. At each step, the *store_name* of the dataset is completed by combining *this_dataset* with the *start_store_name* of that workbook. A custom made form named *select_dataset_form* is completed using the *data_list* function *AddItem* for each dataset. At the end of the *for* loop, the function *Show* presents the user with the form and uses their selection to

finalise the variable *store_name* for that simulation. However, if the user has set *choose_specific* to be *"No"* then a dataset is chosen at random using the *Rnd* function. Once the variable *store_name* has been finalised, the existing 'Weather' spreadsheet in the O&M model's Excel interface is cleared of all data and the *store_name* dataset is copied in its place. This is achieved using the in-built and custom functions; *insert_sheet, num_rows, Select, Copy* and *PasteSpecial*. In the 'Inputs' spreadsheet, the text part of the 'use current weather?' entry is modified to include the name of the new weather dataset.

On the other hand, if the user has selected *use_current_weather* to be *"Yes"* then the existing weather dataset is to be used. An error message is displayed if the user has also selected *choose_specific* to be *"Yes"*. Otherwise, the *store_name* is read from the text part of the 'use current weather?' entry in the 'Inputs' spreadsheet. The left three letters of the existing dataset (*store_name*) must match the acronym for the *site*, and the length of the existing dataset must match the array lifetime. If either condition is not met then an error message is displayed and the user is prompted to exit the program, as described in section 7.2.3.

7.2.7 Statistical run

The subroutine *stat_run_sub* is entered after the user has pressed the 'stat run' button on the 'Inputs' spreadsheet. The time at the beginning of the run is stored in the variable *start_time* using the function *Now*, and *ScreenUpdating* is set to False, in the same way as *run_multi* (section 7.2.2). The progress bar in the bottom left corner of the Excel workbook is updated using the function *StatusBar*. The number of lifetime iterations to undertake (*gen_loops*) is read from the user's selection in the *stat_run_form* shown in Figure 5.1 (page 28). The identifiers *workbook_name* and *data_sheet* are set to be the names of active workbook and spreadsheet respectively, and the number of *years* in the array lifetime is read from the 'Inputs' sheet. The *speed* is set to 1 in order to indicate a 'fast run' process for each of the required loops.

As stated in section 6.3, the user cannot choose a specific dataset of weather conditions for statistical runs. Therefore, an error messaged is produced if the *choose_specific* entry is read as *"Yes"* and the subroutine is ended. The value of *normal_run* is set to False, meaning that all updates of the progress bar (*StatusBar*) need to be made in *stat_run_sub* itself, rather than *run_om*. An array containing the considered statistical parameters, corresponding to the output spreadsheets discussed in section 6.3, is stored in the term *variables*:

- Mean
- Max
- Min
- Range
- Results

The user is then prompted with the message box shown in Figure 5.2 (page 28), asking if they want to start a new series of statistical sheets or continue with the existing ones. The user selected is stored in the *new_results* variable which has the data type *VBMsgBoxResult*. If the user clicks 'Yes' (i.e. if *new_results = vbYes*) then the existing statistical sheets are deleted (*delete_stat_shts*) and new ones are inserted (*insert_sheet* for each *name* in *variables*). The counter than keeps track of the total number of loops undertaken, *num_runs_so_far*, is set to zero. However, if the user clicks 'No' when prompted (i.e. if *new_results = vbNo*) then the 'stat_results' spreadsheet is used to

calculate the value of *num_runs_so_far*. The average values calculated on the far right of the existing 'stat_results' sheet are deleted. If the user clicks anything else then the program ends.

If the statistical sheets have just been created (i.e. if *num_runs_so_far = 0*) then the headers in the 'stat_results' spreadsheet are printed. This includes the parameters 'Availability, Revenue' and 'OPEX' for every year of the array lifetime. It also print the *output_money_format* so the reader can see the units of the monetary values within having to refer to the 'Inputs' spreadsheet. If the statistical runs are continuing from previous ones, then the numerical values on the 'stat_mean' spreadsheet are converted into summations. This function is an important aspect of enabling the user to continue from previous statistical runs with the same input parameters.

The subroutine then enters a *for* loop (with the identifier *i*) where all the requested iterations are considered, starting at the *num_runs_so_far* value plus 1. The *StatusBar* is updated at the start of each new loop to give the user an update of the model's progress in numerical and percentage terms. The main function, *run_om*, is then called to undertake the simulation of that lifetime. If it is the first loop (i.e. if *i = num_runs_so_far + 1*) then the reference IDs of rows in the *results_sheet* containing information about availability (*avail_row*), revenue (*rev_row*), OPEX (*opex_row*) and profit (*profit_row*) are found using the identifier *search_row* and the correct text of the header. This text must match that defined in the *maint_main_output* object (see section 7.24) and the search must start at row 5 to avoid clashing with the 'Availability' breakdown table. Also, if it is the first loop and a new series of statistical runs (i.e. *num_runs_so_far = 0*) then the *results_sheets* is copied into each statistical sheet (named "stat_" plus the term in *variables*). If it is a new series of statistical runs (i.e. if *num_runs_so_far = 0*) and not the first loop (i.e. *if i > num_runs_so_far + 1*), then the sum values (in 'stat_mean') are calculated along with the minimum and maximum in their respectively output sheets. If it isn't a new series of statistical runs then these values are calculated for all loops. The reference ID of the row to print the values to in the 'stat_results' sheet (*this_print_row*) is calculated using the value of *num_runs_so_far*. Printing is then carried out this this sheet using the identified rows of each parameter for each year of the array lifetime, as well as for the average (with profit).

After the loop of iterations, the progress bar is updated to say "Printing results". Statistical parameters are then calculated for the section of the 'stat_results' spreadsheet containing the average annual values (on the far right). This includes the mean, standard deviation and 95% confidence intervals. The percentage can change if required by modifying the constant *z_value* (1.96 is used for 95% confidence). These is achieved by using *WorksheetFunction.Average* and appropriately named variables such as *average_avail*. The total number of simulations, *total_num_runs*, is calculated using *WorksheetFunction.Max* and the loop IDs. Standard deviation is calculated using the typical sample population equation:

$$\sigma = \sqrt{\frac{\sum(x - \mu)^2}{n - 1}}$$

Where σ = standard deviation, x = current value, μ = population mean, n = population size.

The 'stat_mean' spreadsheet is completed by dividing the summed numerical values by *total_num_runs*. The sheet 'stat_range' is filled by using the maximum and minimum values. Each sheet is then subject to the *round_all_deimcals* function for presentation. The output graphs described in section 6.3 are then created by calling *graph_creator.master*. The *StatusBar* is reset

and the user is presented with the time of the process (using *timer*), thereby completing the statistical runs.

7.3 FAILURE PARAMETERS

As stated in section 7.2.1, the *failure_param_list* object is known throughout the VBA code by the variable *fail_param_list*. Figure 7.2 (page 44) shows that the object is first called by the *setup_class* procedure and subsequently creates a new *failure_param* object for each fault category listed in the 'Inputs' spreadsheet. The *failure_param* object of a particular fault category can be identified by any other class module by using calling *fail_param_list* and the function *get_fail_param*.

7.3.1 Start

When the subroutine *start* is called in *fail_param_list*, the total number of fault categories is assigned to the *no_fail* variable. This can be read by other class modules using the function *get_no_fail*. An array of the *failure_param* objects is set up and its size is defined (named *fail_param*) from 1 to *no_fail* using the *ReDim* operator. An identifier, *i*, is then used to consider each fault category in turn and initialises the object by calling *fail_param(i).start*. The reference ID of the row containing the information for each fault category in the *data_sheet* is sent to the *fail_param* as *1+i*. This needs to be changed if the format of the 'Input's spreadsheet is modified.

The subroutine *start* in each *fail_param* object is responsible for reading the relevant information from the *data_sheet* and storing it in the following variables:

- *name* - String, the name of the fault category
- *colour* - Integer, the *ColorIndex* of the ID cell used to define *severity*.
- *power* - Double, the power loss on the ARRAY caused by this fault
- *relevance* - String, determines if the fault relates to the 'Array' or to a 'WEC'
- *percent* - Double, convert the probability of the failure NOT occurring in a year into a probably of not occurring per time step
- *action_reqd* - String, the action required to repair that fault
- *vessel_reqd* - String, the type of vessel required for the repair/marine operation
- *hours_offshore* - Double, the hours required for the repair/marine operation
- *ops_limits_type* - Integer, the ID of the operational limits type required
- *days_onshore* - Variant, the number of days the fault needs at the O&M base to be repaired (reads 'N/A' if onsite repair can be undertaken)
- *part* - Double, the cost of parts for the repair (in £)
- *other* - Double, the value of other costs incurred by the repair (in £)
- *techs_reqd* - Integer, the number of technicians required for the repair
- *severity* - Severity data type (see section 7.1.1), determines the classification of the fault using the *colour*. Note: error handling is in place if the *ColorIndex* does not match a valid entry

Each of these variables can be identified by any other class module by using the *get* functions (e.g. *get_name, get_power* etc.). For example, if a class module wanted to find the name of the fault with the ID '2' then it could call *fail_param_list. get_fail_param(2). get_name* and assign it to a new variable in its own procedure if required. The final line in *start* calls the subroutine *error_finder*.

7.3.2 Error finder

The subroutine *error_finder* in the *failure_param* object is used to identify incompatible selections in the 'Inputs' spreadsheet. The example in the model is that a fault that requires the action 'Retrieve WEC' (*action_reqd*) must have a defined number of days required at the O&M base (*days_onshore*). The custom function *terminate_program* is used to prompt the user to end the simulation. It is likely that the process of tailoring the O&M model to a specific device will produce more incompatible combinations in the table of fault categories. Care must be taken when making selections in the 'Inputs' spreadsheet to avoid such errors.

Note: it is recommended that *failure_param. error_finder* is modified in order to identify all incompatible entries in the input table of fault categories

7.4 MAINTENANCE PARAMETERS

The information pertaining to the scheduled maintenance events, as described in section 4.1.3, is read by the VBA code in the same way as the failure category data (section 7.3). The *maintenance_param_list* object is defined as the *Global* variable *maint_param_list* and is set up by the *setup_class* procedure (in *run_program*). In *maint_param_list*, the number of the scheduled maintenance categories is stored in *no_maint* and can be accessed by other class modules using the *get_no_maint* function. Each maintenance event is assigned a *maintenance_param* object using the variable name *maint_param*. This can be accessed by other class modules with the function *get_maint_param*.

7.4.1 Start

The *start* subroutine in *maintenance_param_list* takes the number of maintenance events and the number of fault categories as arguments from the calling procedure *setup_class*. The number of failure categories is used here to identify the row reference (in the *data_sheet*) of each maintenance event when *maint_param(i).start* is called. Again, this must be modified appropriately if the format of the 'Inputs' spreadsheet is changed.

The *maintenance_param* object operates in very much the same way as *failure_param* (described in section 7.3). Each relevant cell in the 'Inputs' table of maintenance events is read by the VBA code and then stored in variables. One difference from *failure_param* is that *inspection* costs are also included. Again, these costs must be in pounds sterling; the same format as *parts* and *other* costs. There is no *error_finder* procedure in *maintenance_param*, although this could be added by the user if deemed appropriate. Functions can be called by other class modules to *get* any piece of information using the syntax *maint_param_list .maint_param(i).get_name* for example.

7.5 MAINTENANCE MANAGER

The *maint_manager_object* (known in the code simply as *maint_manager*) is the primary control class module of the VBA object oriented program. Aside from the failure and maintenance-related class modules, the *run_program* module does not call any other object directly apart from *maint_manager*. This position of *maint_manager* in the hierarchy of the class modules is shown visually in Figure 7.1 (page 35). The description of the *run_program* object (section 7.2) showed how *maint_manager* is initialised by the *setup_class* procedure (see Figure 7.2, page 44), utilised several times for every time step in each year of the array lifetime, and also controls the printing of key outputs via the *post_process* subroutine. This section describes the procedures in the *maint_manager* object and refers to other sections of this report where relevant.

7.5.1 Defining class modules

At the top of *maint_manager* it is necessary to assign variable names to each class module used throughout the object. The variable names are:

- *weather* - *weather_object*
- *revenue* - *revenue_object*
- *vessel()* - *vessel_object* (note: one object per vessel)
- *parts* - *parts_object*
- *delays* - *delays_object*
- *hindcast* - *hindcast_object*
- *array_object* - *array_object*
- *maint_man_output_arr* - *maint_man_output_list*

In each case, the *Dim As New* terminology is used to define each name as a new object. The other exception is the array of *vessel* objects, where each vessel listed in the 'Vessels' spreadsheet is assigned a new *vessel_object*. It should be noted that the *array_object* name is not shorted in order to avoid confusion between the context of the wave energy array (i.e. a farm of WECs) and a VBA array used to store data. The other variables denoted here for use in the *maint_manager* object are *num_wecs* (i.e. the number of WECs in the array), *run_sh* (i.e. the relevant output sheet for the current year – only used during a 'full run' process), and *ordered_list* (used to prioritise wec repairs by the cost-benefit analysis).

7.5.2 Start

The *start* subroutine in *maint_manager* creates and initialises a number of class modules. First, it sets the *num_wecs* variable to be the *no_total_wecs* argument sent by *setup_class*. The *weather* object is then initialised in order to calculate accessibility of the weather conditions throughout the array lifetime (see section 7.6). An Integer *wecs_per_matrix* is assigned to the number of WECs used to create the power matrix (in the 'Power' spreadsheet) but initialising the *revenue* object. This also uses the weather conditions to calculate the expected revenue earned by the array at each time step (see section 7.7). The *vessel* array is then dimensioned so it can store information about all the vessels listed in the 'Vessels' input spreadsheet (see section 7.8). An additional *vessel* object (*vessel(0)*) is created and is used to store data pertaining to all vessels. The *start* procedure is called for each *vessel*. The *parts* (section 7.9) and *delays* (section 7.10) objects are also initialised here. The *hindcast* object (section 7.11) is only initialised if either of the cost-benefit analysis

options (*CBA_retrieval* or *CBA_onsite*) have been enabled by the user. Finally, *array_object.start* is called in order to initialise all the WEC objects, as well as the array itself.

7.5.3 Insert run sheets

The subroutine *insert_sheet_maint_man* is only called in a 'full run' process is being carried out (i.e. if *speed = 0*), as described in section 7.2.3. It is used to insert all the 'run sheets'; one worksheet for each year of the array lifetime, named 'year1', 'year2', etc. The variable *run_sheet* was previously defined as 'year'. This is combined with each year (*irun*) to store the worksheet name in *run_sh*. The custom function *insert_sheet* is then used to add each sheet in turn and move each one to the end of the workbook.

7.5.4 Determine failure

The sole purpose of the *determine_failure* subroutine in *maint_manager* is to call the same procedure in *array_object*, using *irun* (current year) and *this_interval* (current interval) as arguments. Although this may seem like an unnecessary step, it is fundamental to the object oriented programming structure of the model so that the module *run_program* only ever calls certain class modules through *maint_manager*. The *determine_failure* procedure in *array_object* is used to undertake the Monte Carlo method of simulating when a fault has occurred on the array or on a WEC. This is described in much greater detail in section 7.13.2.

7.5.5 Determine fix

The *determine_fix* subroutine is used to decide whether any scheduled maintenance events are due to take place, and initialises the cost-benefit analysis if appropriate. The same procedure is first called in the *array_object* with *irun* and *this_interval* as the arguments, as seen with *determine_failure*. Then, if either of the cost-benefit analysis options (*CBA_retrieval* or *CBA_onsite*) have been enabled by the user, the *cost_benefit_analysis* object is created and initialised. This is referred to in the subroutine at *cost_ben_analy*. Using *cost_ben_analy*, the full list of WECs that need to be repaired or maintenance is obtained by calling the *create_full_list* function (stored in *full_list).* The *full_list* is then sorted into order of priority by calling *cost_ben_analy.order_this_list*, and stored in the *ordered_list* array. A detailed explanation of the cost-benefit analysis is given in section 7.12. If the cost-benefit analysis is not enabled, then the *ordered_list* is set up to list all the WECs in order of their IDs. The *ordered_list* in used in the *attempt_fix* subroutine of *array_object* so that certain WEC repair or maintenance actions are prioritised if required (see section 7.13.4).

7.5.6 Determine actual fix

The only operation undertaken by *determine_actual_fix* in *maint_manager* is to call *array_object.attempt_fix*, where marine operations and repairs are simulated. (see section 7.13.4). To enable *attempt_fix* to carry out its function, it must be sent the usual arguments defining the current date (*irun* and *this_interval*), as well the variable names of relevant objects (*weather, vessel, parts, delays*) and the *ordered_list* of WECs.

7.5.7 Print interval

The subroutine *print_interval* only runs if *speed = 0* (i.e. if a 'full run' process is being undertaken). In this case, the purpose of *print_interval* in *maint_manager* is to control and format the printing of the 'run sheets' previously detailed in section 6.2.

In order to control the printing of the 'run sheets', *print_interval* identifies the column where each section needs to start printing its information. This is achieving using a series of variables (*date_start_col, array_start_col* etc.) and considering each section in turn, taking the start column of the previous section as a reference point. Certain *get* functions are used to help identify the number of columns in a section (e.g. *get_num_ops_lims_types*). The relevant *start_col* variables are sent to each of the class modules shown in Figure 7.3 in order to print the information to 'run sheets'.

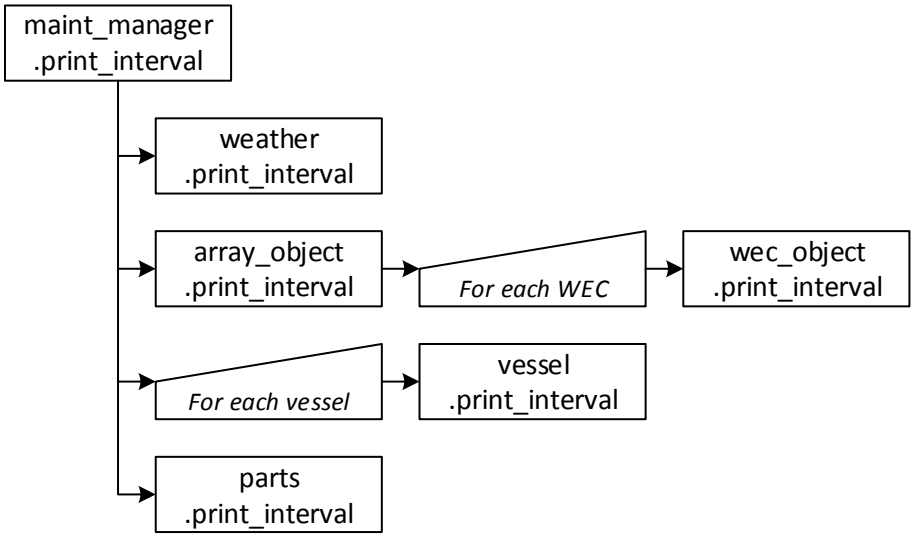

Figure 7.3. Structure of *maint_manager.print_interval*

Once a 'run sheet' has been completed (i.e. when *this_interval = no_intervals*) then it is formatted to produce a user friendly interface. Headers of the various sections are merged together with the text centred. This process uses a *for* loop with a nested *Select Case* to find the correct cells to merge (*this_range*). The WEC and vessel headers are also merged and centred for presentation purposes. Finally, all columns are resized using the *AutoFit* function and the date and header sections are 'frozen' (with *FreezePanes*) as described in 6.2.

7.5.8 Next interval

The *next_interval* subroutine is used to set the simulations up for the following time step. The *maint_manager* object first calls *array_object.next_interval* in order to finalise any completed repairs and make any vessels available again if possible. It is sent the variable names of the appropriate objects as arguments, as well as *num_*wecs. The subroutine *next_interval* in the *parts* object is also called to update the delivery of spare parts, if relevant. These procedures are described in much more detail in sections 7.13.7 and 7.9.7 respectively. Both procedures are sent the current date information, *irun* and *this_interval*, as arguments.

7.5.9 Post process

The bulk of the output information printed to the 'Results' spreadsheet (see section 6.1) is controlled by the *post_process* function in the *maint_manager* object. Throughout the function, the reference ID of the row to start printing (*start_*row) is updated whenever a new class module is called. Three class modules are assigned variables names for use in the *post_process* function:

- *array_output_arr* - *array_output_list*
- *techs_object* - *technicians_object*

The *array_object.post_process* is called in order to produce the full breakdown tables of availability, parts costs, other costs and inspection costs described in section 6.1. The function is described in more detail in section 7.13.13. The procedure is a function because it provides *maint_manager* with the object *array_output_list*. This then used to get the number of parameter tables printed (*get_no_param*) in order to define *start_row* for the next section. The breakdown of revenue is printed by calling *draw* in the *revenue* object. Here, the new row to start printing is calculated within the function *draw*, allowing it to be assigned directly to *start_row*. The *technicians_object* is not defined in *maint_manager*, so *post_process* needs to use the function *array_object.get_technicians_object* to be able to print the output relating to contracted technicians' fees. Again, the value of *start_row* is assigned to the *draw* function to format the 'Results' spreadsheet in a readable manner.

Each vessel in the 'Vessels' spreadsheet is then considered in turn with the identifier *i*. The reference ID of the start row (*vessel_start_row*) is calculated using the number of parameters to print (*get_no_param*). The *post_process* function of each *vessel_output* object is then called in order to print the information. A subroutine is *maint_manager* named *calc_total_vessel_costs* is used to calculate and store the total costs incurred by all vessels. This is achieved by using the zero entry of the *vessel* objects (i.e. *vessel(0)*) and the associated functions *get_vessel_output*. For each year of the array lifetime, as well as the annual average, the vessel costs are summed using the *set* and *get* functions described later in section 7.22. The *start_row* value is updated using the number of printed parameters (*get_no_param*) and the number of vessels (*num_vessels*).

Following the vessels section, the outputs for *delays* are printed by calling *draw*. Again, the new row is calculated within *draw*, allowing it to be assigned to the next *start_row*. The summary table for the array is then set up by calling *maint_man_output_arr.start* (indicating *maint_man_output_list*, as described in section 7.5.1). This needs to read from the relevant objects (e.g. *revenue, techs_object* etc.) to enable key information to be extracted. The subroutine *draw* in *maint_man_output_arr* is then called to print this information to the 'Results' spreadsheet (see section 7.24 for more details). Finally, the number of parameters in the summary table (*get_no_param*) is used to help define the return value of *post_process*, thereby allowing *run_program* to identify the next row for printing the outputs for failures and maintenance, as described in section 7.2.5.

7.6 WEATHER

The *weather_object*, defined in *maint_manager* simply as *weather*, is used to read information from the 'Weather' and 'Ops Limits' spreadsheets and convert the data into a useable format for the VBA code. This means defining 'weather windows' for marine operations (periods of accessibility) at each time step of the array lifetime. The *weather* object also stores the date information for each time step (i.e. month, day and hour). The three variables used throughout *weather* are *num_ops_lims_types* (i.e. the number of operational limits types in 'Ops Limits'), *wndo_array_store* (i.e. weather windows), and *date_array_store* (i.e. date information).

7.6.1 Start

The *start* subroutine of the *weather* object is called during the *setup_class* procedure, shown by the flowchart in Figure 7.2 (page 44). At the top of the subroutine, a number of parameters are defined (*Dim*), including several constant values (*Const*). The following five constant values correspond to specific columns in the input spreadsheets, providing a clear means of editing the code if the format of the sheets is modified:

- *ops_lims_data_col* - column containing all the input data in the 'Ops Limits' sheet
- *year_read_col* - column containing the year ID in the 'Weather' sheet
- *Hs_read_col* - column containing significant wave heights in the 'Weather' sheet
- *T_read_col* - column containing wave periods in the 'Weather' sheet
- *U_read_col* - column containing wind speeds in the 'Weather' sheet

If a new dataset has not been chosen then the 'Weather' dataset will contain printed information about the energy and revenue generated by the array. This information is deleted using the *Range* function *Clear*. The number of operational limits types (*num_ops_lims_types*) is read from the 'Ops Limit's spreadsheet (*ops_limits_sheet*). This variable can be used by other class modules using the function *get_num_ops_lims_types*. The array that is used to store weather window information (*wndo_array_store*) is redefined to be a three dimensional array; *1 To num_ops_lims_types* for each type of operational limits, *1 To no_intervals* for each time step, *1 To no_run* for each year in the array lifetime. The array used to store the date information is also redefined as three dimensional; *1 To 3* to store values of month, day and hour, *1 To no_intervals* and *1 To no_run*. In this regard, both the arrays explicitly store information about every individual time step throughout the array lifetime.

The rest of the *start* subroutine takes place within a *for* loop which considered each of the operational limits types using the identifier *this_lim_type* (from 1 to *num_ops_lims_types*). The temporary variable *this_read_row* is used to find the reference ID of the row (in the 'Ops Limits' sheet) containing the header for that type (*header_row*). The number of parameters that *this_lim_type* considers is read from the appropriate row in the *ops_limits_sheet*. As described in section 4.4, a type with one parameter only considers significant wave height (*Hs_limit*), whilst a type with two parameters considers wind speed (*U_limit*) as well. If three parameters are considered then wind speed is still a constraint, but the significant wave height limit depends on the value of wave period, shown visually with the example in Figure 4.1 (page 24). The relevant input values are read from the *ops_limits_sheet* and stored in appropriately named variables.

The *for* loop then becomes a nested *for* loop by considering each row (*this_read_row*) in the 'Weather' spreadsheet (*weather_sheet*) as well as each limit type. As the *start* subroutine is called

by *setup_class* outside of the year and intervals loop in *run_om* (see section 7.2.3), it must read the year and interval information directly from the *weather_sheet*. The year value (*this_year*) is read using the values of *this_read_row* and *year_read_col*. The interval value (*this_interval*), however, must be calculated based on the row position and *this_year* with the number of intervals per year (*no_intervals*). For this calculation, it is important that the variables are converted to the Long data type (using *CLng*) in order for the equation to work above values of 32,767 (the limit of Integer). The month, day and hour values are placed in *date_array_store* when *this_lim_type = 1* (i.e. so the action is only undertaken once). The reading of these cells is based on each parameter's column position relative to *year_read_col*. At each *this_read_row*, the values for significant wave height, wave period and wind speed are read (using the defined *Const* columns) and placed into the variables *this_hs, this_T* and *this_U* respectively.

If the number of *params_considered* is either 1 or 2, then the value of *this_hs* is assessed against the *Hs_limit*. If the limit is exceeded by the weather conditions (i.e. *this_hs > Hs_limit*) then the weather window is not accessible. In this case, the correct entry in the *wndo_array_store* (i.e. the three dimension positions are *this_lim_type, this_interval* and *this_year*) is made to read "CLOSED". If this limit has not been exceeded then the entry reads "OPEN". Additionally, if *params_considered* = 2 then the *wndo_array_store* entry is set to "CLOSED" if the *U_limit* has been exceeded.

This method of defining weather window is a little more complex if there are three *params_considered*. The entry in *wndo_array_store* is first initialised to read "OPEN". An *If-Else* series of conditions is then utilised to close the weather window if the limits have been exceeded. The wind speed is considered first, with the window "CLOSED" if *this_U* exceeds the *U_limit*. If the wind speed is within the limit, then the lower Hs boundary is considered. If *this_hs* is less than the lower boundary (*lower_max_Hs*) then the window stays "OPEN". If neither of these conditions are met, then the upper boundary of Hs (*upper_max_Hs*) is taken into account. If the boundary is exceeded then the entry of *wndo_array_store* is set to "CLOSED". However, if none of the aforementioned conditions are met, then the limit of significant wave height depends on the value of wave period. To assess this condition, the gradient and y-intercept of the sloped line (demonstrated in Figure 4.1, page 24) must be calculated using following basic line equations. If the wave period value (*this_T*) lies above the sloped line on the graph (i.e. *this_T < ((this_hs – line_y_intercept*) / *line_gradient*) then the *wndo_array_store* entry is set to "CLOSED". An error message will be displayed if an invalid value of *params_considered* has been found.

$$gradient, m = \frac{y_1 - y_2}{x_1 - x_2}$$

$$y.intercept, y_0 = y_1 - (x_1 \times m)$$

If the user has selected the option to limit marine operations to daylight hours only (i.e. if *night_ops_on = False*) then the function *is_daylight* is used to identify whether the current interval is during daylight hours or not. If it is dark during the current interval (i.e. if *is_daylight = False*) then the weather window is set to be "CLOSED".

7.6.2 Get this window

The function *get_this_wndo* is used by other class modules to identify if the weather conditions at a particular time step are accessible ("OPEN") or not ("CLOSED"). For this information to be

obtained, the function must be sent the type of operational limits (*this_lim_type*) and the current interval (*this_interval*) in the present year (*this_year*). The procedure must first identify situations where the requested interval lies in the following year. To avoid returning values as arguments, the values of *this_interval* and *this_year* are assigned to the variables *temp_interval* and *temp_year* respectively. If the interval is greater than the number of intervals in a year (i.e. if *temp_interval > no_intervals*) then the values are updated by adding one to *temp_year* and resetting *temp_interval* appropriately. If the newly updated year is beyond the array lifetime (i.e. if *temp_year > no_run*) then the return value (*get_this_wndo*) is set to "CLOSED". Otherwise, the relevant String value from the *wndo_array_store*, set up by *weather.start*, is returned.

7.6.3 Daylight hours

As stated in section 7.6.1, the function *is_daylight* is used to determine if it is daylight during a particular interval in a given year. To achieve this, it utilises the 'Daylight' spreadsheet detailed in section 4.6. The function is sent *this_interval* and *this_year* by the calling procedure (*e.g. weather.start*) as arguments. The Boolean return value, *temp_bool*, is initialised to True, meaning that it is daylight in the requested interval. The month and hour values are read from the *date_array_store* and assigned to the variables *this_month* and *this_hour* respectively. The reference ID of the row in the 'Daylight' sheet containing header information for selected site (*this_location*) is then selected (*header_read_row*). This value will need to be changed if the format of the spreadsheet is modified. Currently only the 'North Scotland' site is considered but more locations can be added here, as described in section 4.6. The correct row reference for *this_month* is then identified using the *header_read_row* and stored in *daylt_read_row*. The column corresponding to the requested *this_hour* is identified (*daylt_read_col*). Again, these lines of code need to be edited if the format of the 'Daylight' is modified. If the relevant cell in the 'Daylight' sheet reads 'Night', then the return value (*temp_bool*) is set to False, telling the calling procedure that it is dark during the specified interval. The function name, *is_daylight*, must be set to the *temp_bool* variable in order for the calling procedure to recognise it.

7.6.4 Print interval

The *print_interval* subroutine is called by *maint_manager*, as shown in Figure 7.3 (page 52), only when a 'full run' process is taking place. It prints the date information (i.e. month, day, hour) for the current interval and year, as well as the weather window (i.e. accessible or not) for each type of operational limits.

For each new output spreadsheet (*run_sh*) the headers are printed only once using the condition *if this_interval = 1*. This includes the section headers, 'Date' and 'Weather Windows', as well as their respective subsections. Each operational limits type's header is printed using the identifier *i* from 1 to the *num_ops_lims_types*. The date information is then printed to the relevant cells using the *date_array_store* values for the current *this_interval* and *this_year*. Similarly, the information from the *wndo_array_store* is extracted for each operational limits type. If the weather window for that time step is "OPEN" then the cell is filled green using the in-built function *Interior.ColorIndex*. If the window is "CLOSED" then the cell is coloured red.

7.6.5 Longest daylight window

An issue with the option of constraining marine operations to daylight hours has been identified. If a marine operation takes a significantly long time (e.g. a subsea moorings inspection or repair)

then the task might never be carried out if only daylight hours are considered. This is explained further in section 7.13.4. In these situations, it is useful to find the longest period of daylight for the selected array location using the *weather* function *longest_daylight_wndo*.

In *longest_daylight_wndo*, the Integer return value (*temp_int*) is first initialised to be a nominal negative number. As with the *is_daylight* function (section 7.6.3), the *header_read_row* is identified for the selected site (*this_location*). Each month is then considered in turn using the identifier *this_month* in a *for* loop. The number of daylight time steps (*count*) is first initialised to be zero. After the *daylt_read_row* has been identified, a nested *for* loop considers each time step in a 24 hour period with the variable *this_hour*. The relevant row and column (*daylt_read_col*) entries are used to read from each cell in the 'Daylight' spreadsheet and *count* the instances of "Day" for each month. If a new maximum is found (i.e. if *count* > *temp_int*) then the return value is updated. *longest_daylight_wndo* is then set to *temp_int* so the value can be recognised by the calling procedure.

7.7 REVENUE

The *revenue_object* is defined simply as *revenue* by the control class module *maint_manager* (see section 7.5.1). It is used to calculate and store the expected revenue earned by the array throughout its lifetime if it operated constantly at maximum capacity. It also keeps track of the actual revenue generated and controls the printing of information to the output spreadsheets.

A number of variables are defined for use throughout the *revenue* object. The *revenue_interval()* array is used to store the expected revenue earned during every interval of the project lifetime. The output object controlling the printing of contractor fees of every year, *revenue_output*, is defined as *revenue_output_arr()*. The *tariff* is the user-defined sale price of electricity on the 'Power' spreadsheet. The size of the power matrix might change for different types of WEC, so the number of significant wave height and wave periods entries (*num_Hs_entries* and *num_T_entries* respectively) need to be defined. The position of the Hs and T headers in the power matrix also need to be defined. *T_header_row* is identified in the *start* function using the *Const* value *hs_header_col*.

7.7.1 Start

At the beginning of the *start* function in *revenue*, the user-defined *tariff* (in p/kWh) is read from the *power_sheet*, as if the number of WECs accounted for in the power matrix (*wecs_per_matrix*). The total number of WECs is stored in the variable *num_wecs.* The headers used for validating the revenue calculations (average power, energy and revenue) are printed in the *weather_sheet*, to the right of the time series of weather conditions. The value of *T_header_row* is then defined using the custom function *find_index_ref* (section 7.1.16) with the search text "Hs (m)". This code will need to be edited if the layout of the power matrix is modified.

The number of entries of the two parameters, *num_Hs_entries* and *num_T_entries*, is calculated by the custom function *get_num_entries* located at the bottom of the *revenue* object. In each case, the function is told the name of the parameter ("Hs" or "T"), the header position (column or row ID) of that parameter, and the ID of the row or column to start searching. The *get_num_entries* function uses this information to loop through each row (if "Hs") or column (if "T"). The return value, *temp_ret*, is updated at each new cell containing data. The *for* loop is ended when an empty

cell is reached, thereby allowing the function to send *temp_ret* back to *start* as the *num_Hs_entries* or *num_T_entries*. The array *revenue_interval* is re-sized to allow information to be stored for every time step, regardless of the year.

A nested *for* loop then considers each time step (*my_interval*) for every year (*this_year*) of the project lifetime. The identifier of the interval independent of years (*long_interval*) is calculated using this information and *no_intervals* (the number of intervals in one year). The relevant weather conditions (*this_*hs and *this_T*) are read from the *weather_sheet* using their *Const* defined position references (*Hs_read_col* and *T_read_col*). The total number of power matrices required to calculate power generated by the whole array (*num_matrices*) is calculated, assuming a linear relationship between the total number of WECs (*no_total_wecs*) and the number of WECs represented by a single power matrix (*wecs_per_matrix*). The array power for the considered interval, *this_power*, is then calculated using the function *get_power* (see section 7.7.2) multiplied by *num_matrices*. This is converted into energy across the time step (*energy_these_hrs*) before being stored in the correct position in *revenue_interval* (in £). These values are printed to the *weather_sheet* for validation.

After the nested loop, a new output object *revenue_output_arr* is created and initialised (*start*) for every year in the array lifetime. These objects contain the actual revenue earned by the array and will be described in section 7.20. A zero entry (i.e. *revenue_output_arr(0)*) is also created in order to store annual average values. The *revenue.start* procedure is then set to be the *wecs_per_matrix* so that it can be used by the *hindcast* object if required (as described in section 7.5.2).

7.7.2 Get power

The *get_power* function takes values of the two weather parameters, *this_hs* and *this_T*, and finds the corresponding cell in the power matrix in the 'Power' spreadsheet (*power_sheet*). For each parameter, the first cell containing data (*T_first_col* and *Hs_first_row*) is identified using the header reference ID of the other parameter (*hs_header_col* and *T_header_row*). The return value, *temp_power*, is first initialised to be zero. Each wave period entry is then considered in turn using the column reference (*T_first_col*) and the number of entries (*num_T_entries*). If the matching value (*this_T*) is found then each significant wave height value is assessed in turn using *Hs_first_row* and *num_Hs_entries*. When the matching Hs value (*this_hs*) is found then *temp_power* is set to be the power contained in the correct reference cell. The function name *get_power* is set to *temp_power* to be used by the calling procedure.

7.7.3 Get revenue information

The function *get_revenue* can be used by other class modules to obtain the expected revenue earned by the array in a given interval (*my_interval*) and *year*. This date information is converted into an interval value independent of the year (*long_interval*) as described in section 7.7.1. The return value can then be set to the relevant entry in the *revenue_interval()* array. Other class modules can also access the output object (*revenue_output_arr()*) and the user-defined *tariff* by calling the functions *get_revenue_output* (with a year value) and *get_tariff* respectively.

7.7.4 Update revenue

The subroutine *update_rev* takes date information (*irun* and *this_interval*) and the power-generating capacity of the array (*array_power*) to calculate the revenue-related outputs of the

model simulations at each time step. The subroutine interacts with the output object (*revenue_output_arr*) to set the values of earned, theoretical (i.e. expected) and lost revenue (see section 7.20). The subroutine *update_rev* is only called at the end of the *array_object* function *next_interval* (see section 7.13).

7.7.5 Draw

The *draw* function is used to produce the output information for the 'Results' spreadsheet. It is called as part of *post_process* in *maint_manager* (section 7.5.9). The *draw* function is described in much greater detail in section 7.20 (from page 130), along with the following procedures it interacts with in *revenue:*

- *run_title*
- *calc_end*
- *post_process_earned_rev*
- *post_process_theory_rev*
- *post_process_lost_rev*

7.7.6 Revenue estimate

The function *revenue_estimate* is used by the cost-benefit analysis part (section 7.12) of the O&M model in order to estimate the total revenue that could be earned over a specified number of intervals. To achieve this, the calling procedure needs to provide *revenue_estimate* with access to the *hindcast* object, as well as the start and end intervals of the period (*start_int* and *end_int* respectively). These intervals are in a format independent of the year. The cost-benefit analysis part works with WECs only, not the full array, so the power capacity that is sent to *revenue_estimate* is the *wec_power*. This is converted into *array_power* using the number of WECs in the array (*num_wecs*) obtained by the *start* procedure (section 7.7). If necessary, the value of *end_*int is modified so that any intervals beyond the final time step of the array lifetime are not considered.

The return value, *temp_sum*, is then initialised to zero, before a *for* loop considers each time step in the requested period (i.e. from the 'long interval' *start_int* to *end_int*). The function *get_month* is used to identify which month the selected interval is in. It achieves this by converting the 'long interval' into a time step value within a year (*this_interval*). It is not necessary to define the year as they are all the same length (i.e. leap years are not considered). The number of intervals in a day (24 hours) is stored in the variable *ints_day*. The *Select Case* operator is then used to identify the month in which *this_interval* lies. In *revenue_estimate*, the value of *this_month* is sent to the function *get_estimated_monthly_rev* in the *hindcast* object (see section 7.11) in order to get the estimate revenue from the hindcast dataset. The *rev_this_month* is added to the *temp_sum* for every interval after being adjusted for the *array_power* capacity. The function returns to value of *temp_sum* to the calling procedure in the *cost_benefit_analysis* object.

7.8 VESSELS

A *vessel_object* is created for every boat listed in the 'Vessels' spreadsheet (*vessels_sheet*), as shown by the flowchart in Figure 7.2 (page 44). The object is referred to in *maint_manager* by the array variable *vessel,* with the ID of the vessel in brackets (i.e. *vessel(1)*). *Vessel* is used to read information about each boat from the *vessels_sheet* and keep track of its availability and incurred costs throughout the model simulations.

The object responsible for controlling the output of each vessel *(vessel_output_list*) is defined as *vessel_output_arr*. The *state* of the vessel is given the data type *vessel_state*, which means it can either be *not_in_use* or *in_use*, as defined in the functions module (see section 7.1.1). When a marine operation has commenced, the vessel will be *in_use* for a certain number of intervals. This information is stored and updated in the variable *num_ints_left_in_use*. The only *Const* variable defined for the *vessel* object is the number of print columns for each boat (*num_print_cols*). This is used for formatting the 'run sheets' during a 'full run' process. The information related to each vessel listed in the *vessels_sheet* is stored in the following variables:

- *name* - name of the vessel
- *id* - reference ID
- *free_travel_time* - hours from O&M base to site without towing
- *tow_time* - hours from O&M base to site with towing
- *fuel_cost_hr* - the incurred cost for fuel per hour the vessel is in use (in £)
- *personnel_capacity* - the number of technicians allowed on the vessel at any one time
- *day_hire_fee* - the daily hire rate of the vessel (in £)
- *prob_vessel_avail* - the probability of the vessel being available when required

7.8.1 Start

The *start* subroutine for each *vessel* object is used to read all the information for the vessel from the 'Vessels' spreadsheet (section 4.2) for use in the VBA code. It also initialises the output object and common variables for the class module.

The calling procedure of *start* is contained within the *maint_manager* object (see Figure 7.2, page 44). It creates each *vessel.start* and sends it its reference ID as an argument (*i*). As stated in section 7.5.2, a zero entry (i.e. *vessel(0)*) is also created in order to contain "Sum" values. The subroutine *start* only begins to read the information from the *vessel_sheet* is the value of *i* is not zero. The Integer *data_row* is used to define the position (i.e. row reference) of the information for that vessel in the *vessels_sheet*. The code then reads the relevant pieces of information and stores them in the appropriate variable names. Error handling is in place to make sure that the information is being read correctly (i.e. *id* should be the same as *i*). As stated in section 4.2, the cell for *tow_time* will read "N/A" if the vessel cannot be used to tow the WEC to and from site. In this case, the *tow_time* variable is set to zero. An error message is displayed to the user if a vessel capable of towing a WEC has a *personnel_capacity* less than the number of technicians needed for an installation/retrieval operation. This information is read from the relevant cell in the *data_col* of the *data_sheet*.

All vessels have the *state* (with the *vessel_state* data type) initialised to *not_in_use* and the number of intervals left in use (*num_ints_left_in_use*) is set to zero. The output object, *vessel_output_arr*, is initialised by calling the *start* subroutine (see section 7.22.2).

7.8.2 Get functions

The variables within the *vessels* object can be accessed by other class modules by calling the *get* functions (e.g. *get_name, get_num_ints_left_in_use, get_vessel_output* etc.)

7.8.3 Check availability

The Boolean function *check_availability* utilises the user-defined probability of the vessel being available (*prob_vessel_avail*), assigned in the *start* procedure. The in-built function *Randomize* is used in order to generate a truly random number whenever the procedure is called. The return value, *temp_bool*, is first initialised to True. If the random number between 0 and 1 (*rand*) is greater than *prob_vessel_avail* then the return value is changed to False, meaning that the vessel is not available.

7.8.4 Mobilise boat

The subroutine *mobilise_boat* is used to set the vessel *state* to be *in_use*. It also updates the output object by calling the subroutine *add_ints_working* in *vessel_output_arr*. *Mobilise_boat* is sent the number of intervals that the vessel will be in use for (*num_ints*), as well as date information (*irun* and *this_interval*). The value of *num_ints_left_in_use* is set to be *num_ints*.

The subroutine *next_interval* in each *vessel* is used to update the number of intervals the vessel has remaining on the current marine operation. It achieves this by subtracting one from the current value of *num_ints_left_in_use*, regardless of the vessel *state*. An *if* condition is in place to ensure that the value of *num_ints_left_in_use* does not drop below zero.

7.8.5 Demobilise boat

The subroutine *demobilise_boat* is called in order to set the vessel *state* to *no_in_use*. An error message will be displayed if a procedure tries to call *demobilise_boat* whilst the value of *num_ints_left_in_use* is still greater than zero.

7.8.6 Calculate hire fees for an operation

The function *calc_hire_fees_for_op* is used to calculate the total hire fees incurred for a marine operation of a particular length of time. It takes the arguments of the current interval (*this_interval*) and the number of hours of the marine operation (*hrs_transit*). The return value, *total_hire_fees*, is first initialised to zero. As described in section 4.2, vessel hire fees are paid as a flat rate across for any length of time the vessel is used during a 24 hour period (i.e. from midnight to midnight). Therefore, the function *num_new_days* is used to calculate the total number of new days that occurs across the length of the marine operation (*new_days*). This value is then multiplied by the *day_hire_fee* (defined in *start*) to give the *total_hire_fees*.

The function *num_new_days* takes the variables *this_interval* and *hrs_transit* as arguments. The number of *intervals_used* during the operation is calculated by dividing *hrs_transit* by the *time_step* of the model, and using the *WorksheetFunction RoundUp*. The return value, *count*, is initialised to zero ahead of a *for* loop to consider each interval in the marine operation. A line of code accounts for the marine operation crossing over into a new year by checking *if i > no_intervals*. The day at each time step is found by dividing the interval value (*int_now*) by the number of intervals in a 24 hour period (i.e. 24 / *time_step*), and using *RoundUp*. If the interval

under consideration is the first one then *count* is updated to 1. Otherwise, a similar piece of code is used to calculate the day at the previous interval (*prev_day*). If this does not match the current day (*this_day*), then *count* is updated. An error message will be displayed if *count* is not updated at any point.

7.8.7 Calculate fuel costs for an operation

The function *calc_fuel_for_op* takes the argument of *hr_transit* (i.e. length of the marine operation) from the calling procedure and multiplies it by the fuel cost per hour for that vessel (*fuel_cost_hr*) to find the cost of fuel for the marine operation.

7.8.8 Add operation costs

The subroutine *add_op_costs* interacts with both the functions *calc_hire_fees_for_op* (section 7.8.6) and *calc_fuel_for_op* (section 7.8.7) in order to update the vessel output costs for a marine operation. It takes the year value (*irun*) as an argument from the calling procedure so that the output object (*vessel_output_arr*) knows which entry to update (see section 7.22.3).

7.8.9 Print interval

The subroutine *print_interval* is used to print the vessel information for every interval during a 'full run' process, as described in section 6.2. The main header ("Vessels") is printed to the correct cell in the *run_sh* (i.e. spreadsheet named 'year1' etc.) for the first interval of a year, along with the sub-headers of the vessel name, "State", "Hire fees (£)" and "Fuel cost (£)". At each interval, if the vessel is not being used (i.e. *state = not_in_use*) then the correct cell shows this information and is filled green. Otherwise, the cell reads "In use" and is filled red. The cumulative values of hire fees (*cumulative_fees*) and fuel costs (*cumulative_fuel*) are calculated by looping through each year up to and including *irun*, before being printed to the correct cells.

7.8.10 Post process

The subroutine *post_process* is called at the end of the main program in order to control the printing of vessel outputs. The header of the vessels section is printed to the *results_sheet* by calling the *run_title* subroutine (see section 7.22.5) from the output object (*vessel_output_arr*) once using the condition *if id = 1*. The results are printed by calling the *draw* subroutine (see section 7.22.4).

7.9 PARTS

The *parts_object* is known as *parts* by the *maint_manager* object. It is used to control the assigning of spare parts to certain WEC pairs and keeps track of the delivery time when new parts are ordered. Spare parts are only utilised for faults that can be corrected whilst the WEC is on site (i.e. offshore) by replacing the PTO unit or the instrumentation box, for example. The *parts* object is intended to provide a framework for a user who may wish to investigate onsite repairs further. Future modification could see the additional of an input spreadsheet detailing the inventory of spare parts at the O&M base, and could include a more detailed understanding of delivery and lead times.

The variables that are defined for use throughout the object are:

- *type_name()* - an array to store the names of spare parts
- *desired_num()* - the desired number of spare parts of each type to be located at the O&M base at any time
- *current_num* - the number of spare parts of each type located at the O&M base at a particular interval
- *num_types* - the number of types of spare parts
- *delivery_intervals* - the number of intervals to wait for a new delivery
- *num_parts_to_order* - the number of new parts to be ordered
- *wait_time_array* - the time remaining on a delivery of each spare part
- *total_num_spares* - the total desired number of spare parts of all types

7.9.1 Start

The *start* subroutine is called by *maint_manager* as part of the *setup_class* procedure, as shown in Figure 7.2 (page 44). The model considers only replacement of PTO units or instrumentation boxes at present (*num_types = 2*), although this can be modified if required. The arrays *type_name, desired_num* and *current_num* are all re-dimensioned to store information for each part in *num_types*. The variables *num_parts_to_order* and *total_num_spares* are both initialised to zero.

The user-defined input of the 'delivery time (days)' on the 'Inputs' spreadsheet (see section 4.1.1) is multiplied by the number of intervals in a day (i.e. 24 / *time_step*) to set the variable *delivery_intervals*. Each type of spare part is considered in a *for* loop with the identifier *i*. The descriptive cell in the universal inputs table in the *data_sheet* is read as the String *my_str*. In the current model version, this would read 'Number of spare PTO unit' and 'Number of spare instrumentation box'. The in-built function *Mid* is then used to separate the 'Number of spare ' text from the value of *my_str*, thereby identifying the *type_name*. The input value of each type is then stored in the array *desired_num*. The number of spare parts at the O&M base (*total_num_spares*) is updated for each type by adding the value in *desired_num* for each type (i.e. *desired_num(i)*). The value of *current_num* is also initialised to be the user-defined input value.

If the value of *total_num_spares* is greater than zero then the *wait_time_array* is re-dimensioned as a 2-D array, with the first entry being the number of types and the second entry being the total number of spare parts. A nested *for* loop is used to initialise the *wait_time_array*. For each of the *num_types*, every part of that type is considered in turn using the *desired_num* array. The first entry of *wait_time_array* is filled with the ID of that type, whilst the second entry (pertaining to the

wait time on delivery) is set to be zero. Error handling is included to ensure that all entries of *wait_time_array* have been filled successfully.

The number of types of spare parts (*num_types*) can be accessed using the function *get_num_parts_types*.

7.9.2 Order new parts, if available

The subroutine *order_new_parts* takes the arguments of the type of part (*this_type*) and the array of failures on the WEC (*fail_arr*) from the calling procedure. It interacts with two other procedures in the *parts* object, *this_type_id* (see section 7.9.6) and *all_parts_available*, in order to update the correct entries in the *wait_time_array* and keep track of the number of spare parts at the O&M base (*current_num*).

The Boolean function *all_parts_available* is sent the values of *this_type* (String) and *fail_arr* (array of failures). It first sets the return value, *temp_ret*, to False and initialises the counter variables, *count_reqd* and *count_avail*, to zero. An array *temp_current_num* is created to store the *current_num* values (i.e. number of each parts type currently at the O&M base) without the risk of modifying the array itself. The function *correct_type_name* (see section 7.9.5) is then used to double check that the value of *this_type* is a valid type. Each failure in the *fail_arr* is considered using the identifier *i*. The action required to correct the failure is accessed from the *fail_param_list* object (see section 7.3) and should start with the String "Replace" (i.e. identified with the in-built String function *Left*). The rest of the *action_reqd* entry is then read and stored in the variable *read_type*. If the type identified with the fault matches the requested *this_type* then the 'number required' counter (*count_reqd*) is updated. The relevant value in the temporary store array (*temp_current_num*) is checked to see if there is a part of that type available at the O&M base. If so, then the correct value in *temp_current_num* is updated and the 'number available' counter (*count_avail*) is updated. Error handling is in place to ensure that *all_parts_available* is not called when there are no failures. Once all faults in *fail_arr* have been considered, if there is a part of the required type available (i.e. *if count_avail > 0*) then the value of *num_parts_to_order* is set to 1 and the calling procedure (*order_new_parts*) is told of the success (*temp_ret* = True). This feature assumes that only one part of each type is needed per WEC. If this is not the case, then this procedure needs to be modified, with the output conditions accounting for *count_reqd* as well as *count_avail*.

If the value of *all_parts_available* is returned as True, then subroutine *order_new_parts* continues by first identifying the ID of the requested type (*type_id*) by sending the String variable *this_type* to the function *this_type_id*. The current number of parts of the specified type (*current_num(type_id)*) is then updated by subtracting the value of *num_parts_to_order* as defined during *all_parts_available*. In effect, this simulates the site manager assigning a spare part (or parts) to a particular WEC. A nested loop then considers each of the *total_num_spares* at the O&M base (*j*) for each of the *num_parts_to_order* (*i*). If the first entry of *wait_time_array* for the spare part under consideration matches the *type_id* and if that part is not being delivered at the moment (i.e. *wait_time_array(2, j) = 0*), then the second entry of *wait_time_array* is set to be the *delivery_intervals*. This simulates the site manager placing the order for the new parts and then having to wait a certain number of days until they are delivered and ready to be used.

7.9.3 Multiple replacement types

The String function *multi_replacement_types_arr* is sent a list of the failures on a WEC (*fail_arr*) by the calling procedure and is used to return a String list of the types of spare parts needed for those failures.

The variable *count_diff*, used to count the number of different types of spare parts, is initialised to be zero. Each failure in the *fail_arr* is considered in turn, with the String variable *this_type* assigned to be the *action_reqd* to correct that failure (obtaining by interacting with the *fail_param_list* object). If the String value starts with "Replace", then *this_type* is re-assigned to the name of the type of part. If the return array *parts_types_arr* has not yet been created (i.e. *if count_diff = 0*) then *count_diff* becomes 1, and the first entry of *parts_types_arr* is set to contain the String *this_type*. For failures considered after *parts_types_arr* has been created, this assignation is only undertaken if *this_type* is not already an entry in *parts_types_arr*. To check this, the custom function *is_in_array* is used (see section 7.1.9). If the value of *count_diff* is still zero at the end of the failure *for* loop, then the only entry in *parts_types_arr* will read "none". An error message will appear if the function is called with an empty *fail_arr* (i.e. WEC having no failures). The function name, *multi_replacement_types_arr*, is set to be *parts_types_arr* for use by the calling procedure.

7.9.4 Multiple parts types available

The Boolean function *multi_parts_types_available* is used to determine if all required parts from a list of failures (*fail_arr*) are available at the O&M base. The String array variable *parts_types_arr* is set to be the returned value from the function *multi_replacement_types_arr* (see section 7.9.3). If any parts types are identified (i.e. array value is not "none"), then the *count* of the parts available is initialise to zero. A *for* loop then considers each entry in *parts_types_arr* and uses the function *all_parts_available* (described in section 7.9.2) to find if that type is available. If so, then the *count* value is increased by 1. Following the loop, if all parts have been identified as available (i.e. if *count = UBound(parts_types_arr)*) then the Boolean return value is set to True. Otherwise, it returns False. An error message will be displayed if *parts_types_arr = "none"*, due to the fact that this function should only ever be called when relevant, as will be seen later in section 7.14.

7.9.5 Correct type name

The Boolean function *correct_type_name* takes a String value (*this_type*) and checks whether it is the same as one of the parts types. This error handling procedure is in place to ensure that the user doesn't change the descriptive inputs on the *data_sheet* without realising that the *parts* object will be affected. The function utilises the *type_name* array defined in the *start* procedure.

7.9.6 This type ID

The function *this_type_id* takes a String value (*this_type*) and converts it into the ID value (Integer) of the parts type. The function utilises the *type_name* array defined in the *start* procedure. An error message is displayed if the input text (*this_type*) does not match any of the entries in *type_name*.

7.9.7 Next interval

The *next_interval* subroutine is called by the *maint_manager* object, as described in section 7.5.8, in order to set the *parts* object up for the next interval. The subroutine loops for each of the spare

parts in *total_num_spares* (i.e. as defined by the user in the *data_sheet*) using the identifier *i*. The variables *this_type_id* and *this_wait_time* are set to the first and second entries of the *wait_time_ array* respectively. If that parts is currently being delivered (i.e. *if this_wait_time > 0*) then the second entry of *wait_time_ array* is updated accordingly. If this update results in the wait time becoming zero (i.e. the part has just arrived onsite), then the *current_num* entry for that type (*this_type_id*) is updated.

7.9.8 Print interval

The *print_interval* subroutine is called if a 'full run' process is being undertaken (see Figure 7.3, page 52) and is used to print the number of spare parts of each type stored at the O&M base at each interval. If it is the first interval of the year (i.e. *if this_interval = 1*) then the header "Spare parts" and sub-headers with the name of each type (using the *type_name* array) are printed. A *for* loop allows printing of the relevant values from the *current_num* array for each parts type.

7.10 DELAYS

The class module *delays_object* is referred to as simply *delays* by the control object *maint_ manager.* It is used to update the causes of delays to marine operations or other work throughout the lifetime of the WEC array. The corresponding output object, *delays_output*, is accessed with the variable *delays_output_arr*.

The *delays* object and output are structured in a similar way to *revenue* (section 7.7), whereby the main class module (i.e. *delays_object*) acts as the output control object, as well as the main source of functionality. This is a slight difference to other sections of the model, such as the *vessels* object, where the responsibility for output control is assumed by the *vessel_output_list*, rather than the *vessels_object* itself.

Only two procedures in *delays* are described in this section; *start* and *add_this_delay*. The rest of the class module is discussed in section 7.23.

7.10.1 Start

The *start* subroutine in *delays* is called during the key procedure *setup_class*, as shown in Figure 7.2 (page 44). It is used purely to create and set up the output objects for the causes of delays. A new *delays_output_arr* object is created and initialised (by calling *start*, see section 7.23) for each year of the project lifetime (*no_run*). The zero entry (i.e. *delays_output_arr(0)*) is used to store the annual average values.

7.10.2 Add this delay

The subroutine *add_this_delay* is called whenever a marine operation or any other work is delayed. It is sent the current year (*irun*) and the cause of the delay (*delay_type*) by the calling function. For the relevant output object (i.e. *delays_output_arr(irun)*), the *work_attempted* value is updated, together with the specified cause of the delay. If the cause is not recognised then an error message is displayed. More detailed on how this information is used by the VBA code is given in section 7.23.

7.11 HINDCAST

The class module *hindcast_object* is referred to as *hindcast* by the *maint_manager* (see section 7.5.1). The *hindcast* object is only created if either of the cost-benefit analysis options (*CBA_retrieval* or *CBA_onsite*) have been enabled by the user (see section 4.1.1). It is used to convert the hindcast time series of weather conditions stored in the "Hindcast" spreadsheet into estimates for monthly revenue generated by the array and wait times prior to installation of a WEC. Although the accuracy of weather forecasting has improved greatly in recent years, it is likely that hindcast datasets will still play a part in decision making for offshore operations and maintenance for many years to come. Incorporating such analysis into the O&M model could provide an additional level of realism to the functionality of the tool. The code stored in *hindcast* is very similar to the *weather* object (section 7.6). Throughout this section, only the key differences from the *weather* object are discussed in detail.

The primary information calculated by the *hindcast* object is stored in two arrays with the Double data type:

- *estimated_monthly_rev* - estimated revenue earned by the array (if at 100% capacity) in each month
- *estimated_months_install_wait* - estimated number of intervals to wait for a weather window suitable for WEC installation in each month

The name of the 'Hindcast' spreadsheet is defined for the object as *hindcast_sheet*. Constant values (*Const*) are set for the reference IDs of the columns in the *hindcast_sheet* containing the month (*month_col*), hour (*hour_col*), significant wave height (*Hs_col*), wave period (*T_col*) and wind speed (*U_col*). The column containing the key data in the 'Ops Limits' spreadsheet (*ops_limits_sheet*) is also assigned a *Const* value (*ops_lims_data_col*). The user-defined type of operational limits for a WEC installation (in the *data_sheet*, see section 4.1.1) is stored in the variable *install_wndo_type*. Information extracted from the relevant section in the *ops_limits_sheet* (identified by the *install_header_row*) is stored in the following variables:

- *params_considered*
- *Hs_limit*
- *U_limit*
- *lower_max_Hs*
- *upper_max_Hs*
- *lower_T*
- *upper_T*
- *line_gradient* - calculated
- *line_y_intercept* - calculated

7.11.1 Start

As shown in Figure 7.2 (page 44), the *start* subroutine is called by *maint_manager* but only if the cost-benefit analysis (CBA) is enabled. This means that one or both of the Boolean variables *CBA_retrieval* or *CBA_onsite* must be set to True (see section 4.1.1). The *start* subroutine is used to read the information stored in the *hindcast_sheet* and calculate the estimated monthly revenue and wait times.

The *count_matches* array, used to store the number of intervals in the dataset for each month, is first re-dimensioned (i.e. 1 to 12). The *estimated_monthly_rev* array is also re-dimensioned as this size in order to store a single value for each month. The *estimated_months_install_wait* array however, is created to be a 2-D array, with only the first dimension sized 1 to 12 (for each month). The second dimension of *estimated_months_install_wait* is sized as 1 to 16. Here, the first 8 entries correspond to the number of intervals in the required weather window where marine operations can be undertaken at night (i.e. *night_ops_on*, section 7.2.1). For example, if the resolution (*time_step*) of the model was 3 hours, then the second entry of the first month (i.e. referred to as *estimated_months_install_wait(1,2)*) is the number of intervals to wait for a 6 hour (2 x 3 hours) installation weather window in that month. The second set of 8 values in the second dimension (i.e. 9 to 16) store the number of intervals in the required weather window where marine operations cannot be undertaken at night (i.e. restricted to daylight hours only). The maximum of 8 intervals in a row has been used in the model due to the likelihood that a resolution of three hours will be used (and the unlikelihood that a marine operation will take over three hours).

Note: If the CBA is enabled, and if either the model resolution is increased (say to 1 hour time steps) or a marine operation may take over 24 hours, then the size of the second dimension of *estimated_months_install_wait* will need to be increased, and the code changed accordingly.

The number of rows in the *hindcast_sheet* (*num_rows*) is calculated using the custom function *num_rows* (see section 7.1.11). In order to avoid complications later in the code, the name of the active sheet is stored in the variable *sht_to_activate* before the *hindcast_sheet* is selected, thereby allowing the original sheet to re-activated at the end of the procedure. The type of operational limits for WEC installation (*install_wndo_type*) is read from the relevant cell in the *data_sheet*. This information is used to find the reference ID of the row in *ops_limits_sheet* containing the header for that type (*install_header_row*). The value of *params_considered* and the relevant variables pertaining to the weather constraints (e.g. *Hs_limit* etc.) are obtained by reading from the *ops_limits_sheet* in exactly the same way as the *weather.start* procedure (see section 7.6.1).

Also similarly to *weather.start*, the code then enters a *for* loop to consider each row in the *hindcast_sheet*. The values of *this_month, this_hour, this_hs* and *this_T* are read from the defined columns. The code only continues if the values of significant wave height (*this_hs*) and wave period *(this_T*) are numeric, using the in-built function *IsNumber*, thereby avoiding errors due to cells containing headers or NaN entries (Not a Number). The relevant entry in the *count_matches* array (i.e. *count_matches (this_month)*) is added to for each row. The values of Hs (this_hs) and T (this_T) are placed into 'bins' in order to correspond to the power matrix entries (see section 4.5) using the custom function *rounded_val* described in section 7.11.2. These 'binned' values are then sent to the *get_power* function in the *revenue* object (section 7.7.2) in order to find the corresponding value from the power matrix. This is divided by the value of *wecs_per_matrix* (sent to *hindcast* from *revenue* via *maint_manager*, see section 7.7.1) in order to calculate the hourly power (*this_power*) generated by the array (at 100% capacity). The relevant monthly entry in the *estimated_monthly_rev* array is updated by adding the revenue earned in that time step:

$$revenue\ (£)\ per\ time\ step = power\ (kWh) \times time\ step\ (hrs) \times \frac{tariff\ (p/kWh)}{100}$$

Where the tariff is obtained using the *get_tariff* function in the *revenue* object. A nested *for* loop then considers each of the entries in the *estimated_months_install_wait* array with the identifier *num_intervals_wndo* looping from 1 to 8 (i.e. considering half the entries because the first half is if

night_ops_on = True, and the second half is daylight hours only). The Boolean function *this_wndo_open* (see section 7.11.3) identifies if the required length of weather window (*num_intervals_wndo*) is accessible given the installation weather conditions (*install_wndo_type*), assuming operations can be undertaken at night. If *this_wndo_open* is True (i.e. the window is accessible) then the relevant entry in the *estimated_months_install_wait* 2D array (i.e. *estimated_months_install_wait (this_month, num_intervals_wndo)*) is updated (i.e. add 1). If the user-defined variable *night_ops_on* has been set to False then marine operations are constrained to daylight hours only, as described in section 7.6.3. In this case, the other half of the second dimension of the *estimated_months_install_wait* array (i.e. 9 to 16) is also updated (i.e. add 1). This is achieved by using the Boolean function *this_daylight_wndo_open* described in section 7.11.4. Note that the relevant entry of the array is identified using *num_intervals_wndo* plus 8 (i.e. *estimated_months_install_wait(this_month, num_intervals_wndo + 8)*). The nested loop therefore fills the two arrays (*estimated_monthly_rev* and *estimated_months_install_wait*) with the summed values of revenue and the number of open weather windows respectively for each month.

After every row in the *hindcast_sheet* has been assessed, a new *for* loop is entered where each month is considered in turn (i.e. *i* from 1 to 12). The average revenue for that month is calculated by dividing the relevant entry in *estimated_monthly_rev* by *count_matches*. The variable *max_val* is used to identify the last filled entry of the second dimension in *estimated_months_install_wait*. In other words, *max_val* is 8 if *night_ops_on* has been set to True because the remaining half of the array does not get completed if marine operations are not constrained by daylight hours (see previous paragraph). A nested *for* loop then considers each of these entries (i.e. up to *max_val*) using the identifier *num_intervals_wndo*. The relevant entry in the *estimated_months_install_wait* array is first changed to be the probability of the weather window being accessible (i.e. *estimated_months_install_wait(i, num_intervals_wndo) / count_matches(i)*). If the entry is zero (i.e. there are no accessible windows of that length (*num_intervals_wndo*) in that month (*i*) then it is modified to give a very small probability (0.001 currently used). The following equation is then used to convert the probability of the window being open (P_{open}) into estimated wait times (in intervals), as detailed by Gray (2017):

$$wait\ time = \frac{1}{P_{open}} - 1$$

The subroutine ends by reactivating the original sheet (*sht_to_activate*).

7.11.2 Rounded value

The Double function *rounded_val* is called by *start* to convert the hindcast values of significant wave height and wave period into the binned values corresponding to the power matrix in the 'Power' spreadsheet, as described in section 7.11.1. The function is sent the name of the parameter to round (*this_type*) as a String value containing either "Hs" (significant wave height) or "T" (wave period), as well as the numerical value (*this_value*). The reference ID of the row in the *hindcast_sheet* (*i*) is also sent to the function. An *if-else* condition is used to select the correct value of *this_type*. The Double variable *temp_val* is used to store updated values at each step of the 'binning' process. First, *this_val* is divided by the size of the parameter step (i.e. in this example, 0.5m for Hs and 2s for T). This is rounded up to the nearest integer (using *RoundUp*) and multiplied by the parameter step to place the *temp_val* at the upper end of the relevant 'bin'. Half the parameter step is then subtracted in order to place *temp_val* at the centre of the 'bin' (e.g. *temp_*

val = temp_val – 0.25 for Hs). Upper and lower limits are also applied, along with error handling in the case of a negative value being identified. This method of 'binning' the values is exactly the same as seen when generating new time series' using the Markov Chain Method, described in the 'Weather Simulation Report' (WES, 2017a). The function name, *rounded_val*, is set to *temp_val* in order to be read by the calling procedure.

Note: if the parameters steps in the power matrix are modified, then the generated time series' of weather conditions must be changed accordingly, as must the code in *hindcast.rounded_val*

7.11.3 This window open

The Boolean function *this_wndo_open* is called by the *start* procedure (see section 7.11.1) in order to determine if a weather window of a certain number of intervals in length is accessible for installation of a WEC. It is sent the reference ID of the row of the assessed interval (*this_row*) and the required length of the weather window (*this_num_ints*). The return variable, *temp_bool*, is first initialised to True, saying that the weather window is open (i.e. accessible). A *for* loop then considers the weather conditions in each row of the required window (i.e. from *this_row* to *this_row + this_num_ints -1*) and uses the function *int_wndo_open* to 'close' the window (i.e. set the return value to False) if the conditions at any of the intervals are inaccessible. If the value of *this_hs* is a NaN (Not a Number) then the requested row must be beyond the last dataset entry and the function returns False.

The function *int_wndo_open* is used to analyse the weather conditions of a particular row in the *hindcast_sheet* and determine if that interval is accessible for the marine operation defined in *start* (where the variables *params_considered*, *Hs_limit* etc. were read from the *ops_limits_sheet*, see section 7.11.1). The function follows exactly the same structure as the *start* procedure in the *weather* object (section 7.6.1), where the relevant weather constraints are assessed and the weather window is set to open (i.e. *int_wndo_open = True*) if accessible, or closed if not (i.e. *int_wndo_open = False*).

7.11.4 This daylight window open

The Boolean function *is_daylight_wndo_open* also interacts with the function *int_wndo_open* (described in section 7.11.3) in order to assess the accessibility of a weather window of a given length (*this_num_ints*) if operations are constrained to daylight hours only. As with the function *this_wndo_open* (section 7.11.3), the reference ID of the row in the *hindcast_sheet* is sent as *this_row*. Each row in the weather window is considered and the date values of *this_month* and *this_hour* are obtained. These variables are sent to the Boolean function *is_daylight* which utilises the daylight hours matrix (shown on the 'Daylight' input spreadsheet) for the user-defined site in exactly the same way as the *weather* object (see section 7.6.3). Again, the *is_daylight* function in the *hindcast* object needs to be modified if a site other than "North Scotland" is assessed. If it is not daylight then the weather window is closed. Otherwise, the function *int_wndo_open* (see section 7.11.3) is used to close the weather window if the conditions are inaccessible. As before, the return value is set to False if the row under consideration goes beyond the last row of the dataset.

7.11.5 Get functions

The information calculated by the *hindcast* object can be accessed by other class modules by using the functions *get_estimated_months_install_wait* (with the arguments *this_month* and *size_of_window*) and *get_estimated_monthly_rev* (with *this_month*).

7.12 COST-BENEFIT ANALYSIS

The *cost_benefit_analysis* object is utilised at every interval throughout each year of the simulated array lifetime during the *determine_fix* subroutine in *maint_manager*, as described in section 7.5.5. The object is only used if either (or both) of the user-defined options *CBA_retrieval* or CBA_onsite have been enabled (see section 7.2.4). The cost-benefit analysis (CBA) adds a significant amount of time onto the model simulations, which is why the object is only utilised if the user has specifically requested it. The flowchart in Figure 7.4 shows how the *cost_benefit_analysis* object is structured and provides a visual representation of its purpose and functionality.

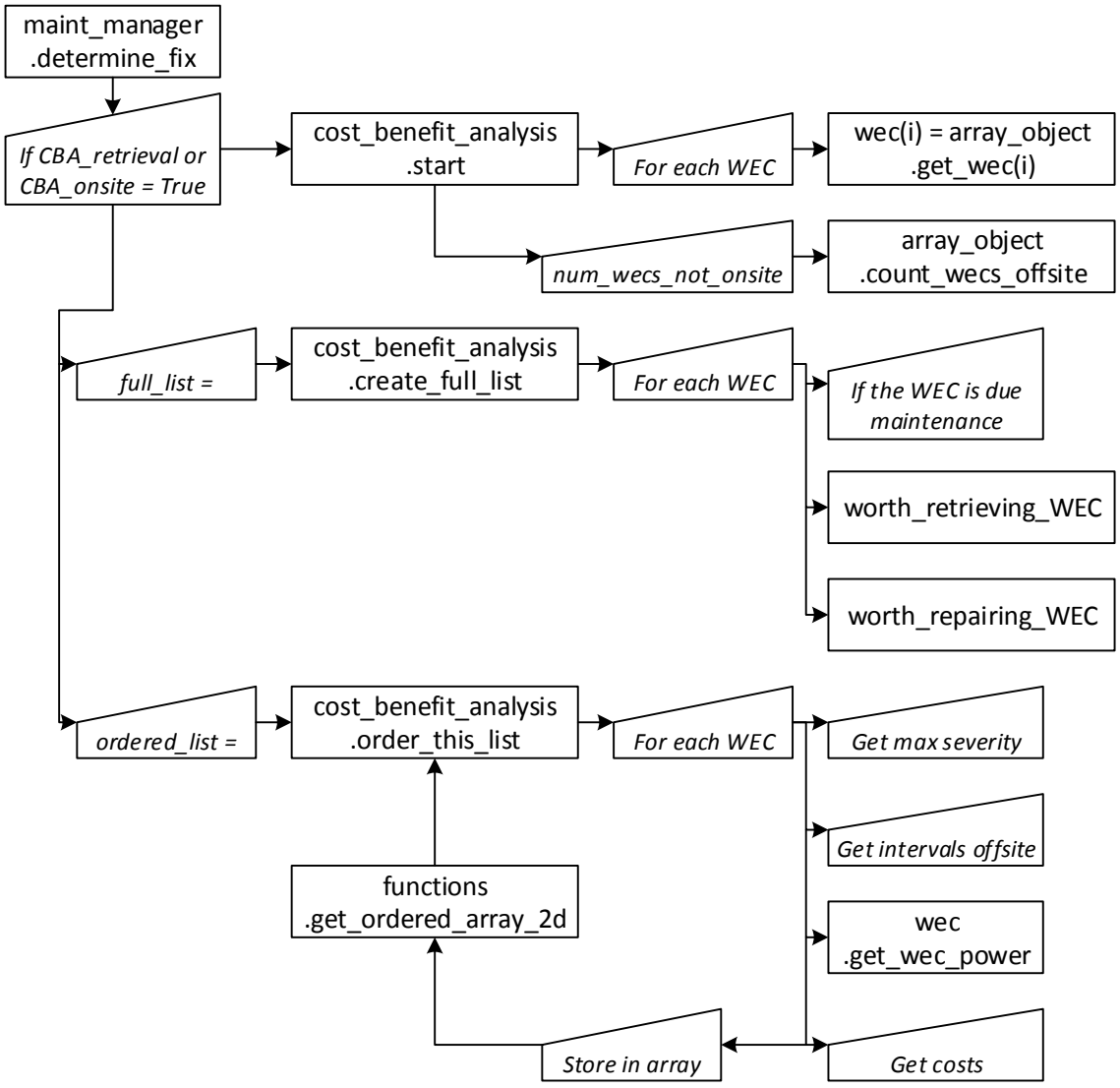

Figure 7.4. Structure of the cost-benefit analysis object

Figure 7.4 shows how the object replicates each of the *wec_object* class modules in order to obtain information about each WEC in the array (stored in *no_wecs*) at every interval. These objects are stored using the variable *wec()*. The interval (in the year) when the CBA object is called is defined as *this_interval*, which then gets converted into an interval value independent of the year (*current_interval_lng*). The number of WECs not onsite (i.e. either in transit or at the O&M base) is stored in the variable *num_wecs_not_onsite*.

7.12.1 Start

The *start* subroutine is sent a number of variables as arguments by the calling procedure (*maint_manager.determine_fix*). The date information is contained in the *irun* (i.e. current year) and *this_interval* (i.e. current interval in *irun*). The *array_object* class module is obtained so that it can be used by the CBA, as well as the total number of WECs in the wave energy array (*num_wecs*). The *start* subroutine is used to set *cost_benefit_analysis* up by assigning the object-based variables *no_wecs* and *interval_now* to be *num_wecs* and *this_interval* respectively. It also loops through each WEC in *no_wecs* (i.e. *wec(i)*) and stores the relevant class module using the *get_wec* function in the *array_object* (see section 7.13.14). The interval independent of the year (*current_interval_lng*) is calculated using *irun*, *this_interval* and *no_intervals* (the number of intervals in a year). The variable *num_wecs_not_onsite* is defined using the *count_wecs_offsite* function in *array_object* (see section 7.13.3).

7.12.2 Create full list

The function *create_full_list* is used to compile an array containing the IDs of any WECs that have been set for repair. It is called by the *determine_fix* procedure in *maint_manager* (as shown by Figure 7.4, page 71) with the arguments *irun* (current year), *vessel* (the *vessel_object* class module), *hindcast* (the *hindcast_object* class module) and *revenue* (the *revenue_object* class module).

Each WEC in the wave farm is considered in a *for* loop using the identifier *i* up to *no_wecs*. The WEC is only considered further if its *state* (obtained by using *wec(i).get_state*) is *on_site* (see the custom data type *WEC_state* in section 7.1). If the WEC is due to undergo scheduled maintenance then it is added to the *full_list*. This information is obtained using the function *any_maint_ready* in the *wec_object* (see section 7.14.34). Otherwise, the list of failures on the WEC is stored in the *fail_list* variables by obtaining the *wec_fail_list* class module (with *wec(i).get_fail_list*) and using the function *get_fail_arr_id* (see section 7.16.3). If there are any failures (identified by checking that the first entry in *fail_list* is greater than zero) then the *wec* function *any_fails_need_retrieval* (see section 7.14.6) is used to identify if the action required to fix any of the faults listed in *fail_list* is "Retrieve WEC". In this case, the WEC is added to the *full_list* if the CBA procedure *worth_retrieving_WEC* (see section 7.12.3) returns True. If the WEC has suffered failures but none require the WEC to be retrieved, then the CBA function *worth_repairing_WEC* (see section 7.12.4) is used to determine if the device ID should be added to the *full_list*. Throughout the *for* loop, the variable *num_in_list* is used to keep track of the number of WECs that have been added to the *full_list*. If *num_in_list* is still zero at the end of the *for* loop, then the first entry in the *full_list* is given a nominal negative value (-5) to tell other procedures that no WECs have been set for repair. The function name, *create_full_list*, is set to be the *full_list* so it can be returned to the calling procedure, *determine_fix*.

7.12.3 Worth retrieving WEC

The Boolean function *worth_retrieving_WEC* is called by *create_full_list* (see section 7.12.2) in order to assess whether or not the WEC under consideration should be set for retrieval (to be repaired at the O&M base) at the current interval, or if it should be left onsite operating with one or more faults until it is next due to undergo scheduled maintenance. Figure 7.5 shows the structure and functionality of the *worth_retrieving_WEC* function in flowchart form.

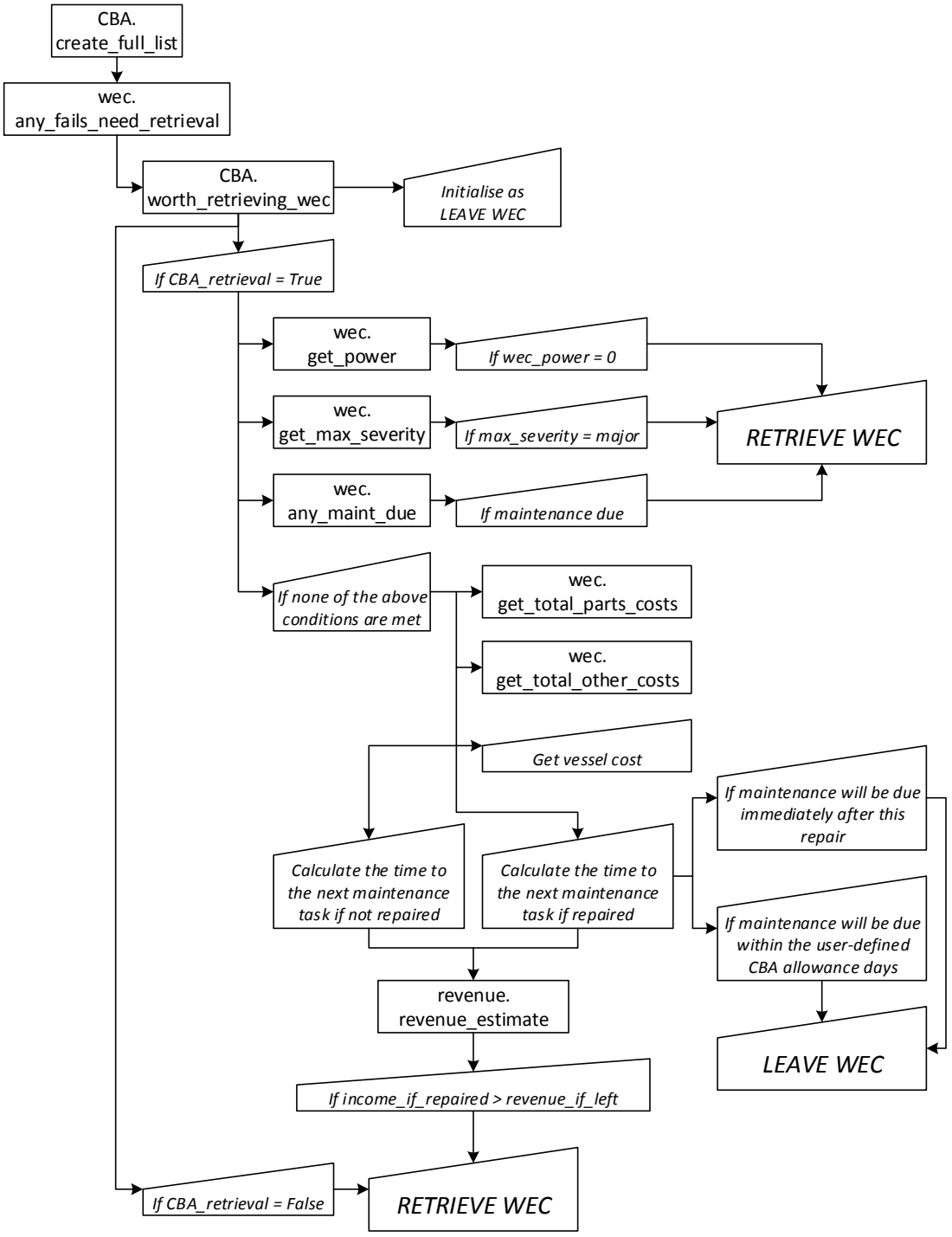

Figure 7.5. Structure of the *worth_retrieving_WEC* function

To carry out its operation, *worth_retrieving_WEC* is sent the variables *irun* (the current year) *wec_id* and *fail_list* (list of faults currently sustained by the WEC), as well as the class modules *vessel, hindcast* and *revenue*. A large number of variable names are used throughout the function, such as the return value *temp_bool*. These are discussed throughout this section where relevant. Note; where procedure names have been made clear in Figure 7.5, these are not discussed further in the following text. The procedures called from the *wec_object* class module are described in section 7.14.

After the retrieval conditions have been assessed (e.g. *wec_power = 0*), the code calculates the *parts_costs* and *other_costs* for the faults listed in the *fail_list*. The *vessel_cost* is not obtained in as direct a manner as these costs. Instead, it is first initialised to zero before being updated by other procedures called throughout the remaining code. The *wec* function *get_time_until_repaired* (see section 7.14.29) calculates the number of intervals from the time the vessel is set for the retrieval operation until the WEC is ready to be installed following offsite repair (*ints_not_onsite*). It also updates the *vessel_cost* value with sum of the hires fees and fuel costs incurred for the retrieval operation. Using the calculated *ints_not_onsite* and the interval independent of the year (*current_interval_lng*) as arguments, the *revenue* function *get_month* (see section 7.7.6) obtains the ID of the month where the WEC installation operation will be attempted (*month_to_install*). The length of the installation operation in intervals (*length_install*) is found using the *wec* function *get_install_time* (see section 7.14.30). The *vessel_cost* value is again updated for the costs incurred during that marine operation. The *hindcast* function *get_estimated_months_install_wait* then obtains the number of intervals required to wait for an accessible weather window (given the installation weather constraints and *length_install*) using the hindcast weather dataset (see section 7.11) and stores it in the variable *wait_time_install*. The number of intervals the WEC is not onsite (*ints_not_onsite*) is updated by adding the *length_install* and *wait_time_install*. Using this information, the *wec* function *ints_to_next_maint* then calculates the number of intervals until the next scheduled maintenance event if the WEC is repaired (*ints_to_maint_after_repair*) and if the WEC is left on site (*ints_to_maint_if_left*).

If the number of intervals until the next scheduled maintenance event if the WEC is repaired (*ints_to_maint_after_repair*) is returned as -5, then the *wec* function *ints_to_next_maint* has identified that the event will be due immediately after the repair. In this case, the return value *temp_ret* is set to False, indicating that the retrieval operation should be delayed until the maintenance event is due. This also happens if the value of *ints_to_maint_after_repair* is less than or equal to the number of CBA allowance days (converted to intervals) specified by the user in the *data_sheet* (*CBA_allowance_days*, see section 4.1.1).

If neither of the above conditions is met, then a cost-benefit analysis is undertaken to compare the revenue if the WEC is left at site (perhaps operating at reduced capacity) against the total income if the device is repaired (i.e. revenue at full capacity after repair minus repair costs). The estimated revenue is each case is obtained from the *revenue* function *revenue_estimate* (see section 7.7.6), which itself interacts with the *hindcast* object (section 7.11), using the appropriate interval values and *wec_power*. If the *income_if_repaired* is greater than the *revenue_if_left* then the return value *temp_ret* is set to True, thereby setting the WEC for retrieval. If *CBA_retrieval* is set to False (which must mean that the CBA only runs for onsite repairs), then the WEC is set to be retrieved as soon as possible in line with the default functionality of the model. The calling procedure (*create_full_list*) is told of the decision by setting its name to *temp_ret*.

7.12.4 Worth repairing WEC

The function *worth_repairing_WEC* is called by *create_full_list* if none of the WEC's sustained failures require the device to be retrieved for onsite repair (as shown in Figure 7.4). It is used to determine if the faults should be corrected (i.e. have certain parts replaced whilst the WEC is onsite) or if the WEC should remain onsite operating at reduced capacity. The structure and functionality of *worth_repairing_WEC* is almost identical to that of the function *worth_retrieving_WEC,* discussed in detail in section 7.12.3 and shown visually in Figure 7.5 (page 73). One key difference is that all instances of *CBA_retrieval* are replaced by *CBA_onsite*. In addition, it is not necessary for the function to consider the estimated wait times (calculated by the *hindcast* object, section 7.11) because an installation operation is not due to be carried out. Instead, the relevant section of the *wec* function *get_time_until_repaired* is used, as shown in section 7.14.29, in order to calculate the number of intervals required to complete the offshore repairs.

7.12.5 Order this list

The function *order_this_list* is called by the *determine_fix* procedure in *maint_manager*, as described in section 7.5.5 and shown visually in Figure 7.4 (page 71). It is used to sort the list of WEC IDs determined during the cost-benefit analysis process in *create_full_list* (see section 7.12.2) into a particular order. The function is sent the list of WECs set for repair or maintenance (*this_list*) from the calling procedure, as well as the relevant class modules (*vessel, revenue, hindcast*). A number of new variables are used throughout the *order_this_list* function and are discussed throughout this section.

The number of WECs in *this_list* is stored in the variable *count_wecs*. A Variant array, *array_long*, is re-sized as a 2-D array where the first dimension is from 1 to *count_wecs* and the second dimension is from 1 to the number of parameters used to order to the list (7 in the model example). If there is only one WEC in the list (i.e. *count_wecs = 1*) then the return value *order_list* (a 1D array) is set to equal *this_list*. If *count_wecs* is greater than 1 then each of the WECs in *this_list* (*wec_id*) is considered in turn (using the identifier *i*). The list of failures occurred on each WEC is stored in the variable *fail_arr* by using *wec_fail_list* (see section 7.16.3). The first four entries of the second dimension of the Variant *array_long* are then filled with the following information for each WEC (i.e. *array_long(i, 1), array_long(i, 2)* etc.):

1. *wec_id* - ID of the WEC (*this_list(i)*)
2. *major_failures* (0 for True) -*wec(wec_id).major_failures(fail_arr)* – section 7.14.28
3. *any_maint_ready* (0 for True) - wec(wec_id).any_maint_ready – section 7.14.34
4. *intermediate_failures* (0 for True) - *wec(wec_id).intermediate_failures(fail_arr)* – section 7.14.28

If the conditions are met (e.g. if the WEC has a major failure) then the relevant entry is given the value 0. Otherwise, it is given 1. This format is in place because the custom *function get_ordered_array_2d* (see section 7.1.12) orders the array with the smallest values first.

The last three entries (i.e. *array_long(i, 5)* to *array_long(i, 7)*) contain the values of the number of intervals the WEC requires offsite, the power capacity of the WEC (*wec(wec_id).get_wec_power*), and the total costs of repair (including vessel costs) respectively. If the WEC is either due to undergo maintenance or onsite repairs (i.e. replacing parts offshore), however, then these entries are filled with zeros. This is justified because the severity of the failures is enough information to sort WECs with onsite repairs only. If the WEC is to undergo a retrieval operation to repair failures

then the total number of intervals the WEC will not be onsite (*ints_not_onsite*) is calculated in exactly the same way as seen in the *worth_retrieving_WEC* function (section 7.12.3). The process utilises functions in the objects *wec, revenue, vessel* and *hindcast* and also obtains the total *vessel_cost* incurred during both the retrieval and installation operations. The *parts_costs* and *other_costs* for the repairs are obtained from the *wec* object (see section 7.14.26).

Once each WEC in *this_list* has been considered, the completed *array_long* variable is sent to the custom *function get_ordered_array_2d* (see section 7.1.12) to re-order the array in the following hierarchy. Note: this hierarchy can be modified by the user if required.

1. Suffered major failure/s
2. Due scheduled maintenance
3. Suffered intermediate failure/s
4. Fewest intervals not onsite
5. Lowest power capacity
6. Lowest total cost to repair

The ordered version of *array_long* is stored in the variable *sorted_array*. The return variable *order_list* is re-sized to be a 1-dimensional array from 1 to *count_wecs*. For each WEC, the relevant value in *order_list* (i.e. *order_list(i)*) is filled with the WEC ID identified in the *sorted_array* (i.e. *sorted_array(i, 1)*). Finally, the function name *order_this_list* is set to be *order_list* so it can be recognised by the calling procedure *determine_fix* (section 7.5.5).

7.13 ARRAY OBJECT

The *array_object* class module is used to create, set up and control a new *wec_object* for each of the WECs in the wave energy array. It is known as *array_object* in the control object *maint_manager* (see section 7.5.1) rather than *array* (as is the format for other class modules, e.g. *revenue*) in order to avoid confusion with a VBA array. The *array_object* also controls the functionality of the model for failures and maintenance events that are related to the entire wave energy array. These categories are identified on the 'Inputs' spreadsheet by having the *relevance* entry set to "Array", as described in sections 4.1, 7.3 and 7.4.

The variables defined for use throughout the object are as follows:

- *state* - the current state of the array (data type *array_state*, section 7.1.1)
- *wec()* - a class module *wec_object* for each WEC (section 7.14)
- *num_wecs* - the total number of WECs in the array
- *array_fail_arr* - the class module *array_fail_list* (section 7.16.1)
- *array_output_arr* - the class module *array_output_list* (section 7.17)
- *array_power* - power capacity of the array (between 0 and 1)
- *repair_intervals* - the number of intervals remaining to complete array repairs
- *technicians* - the class module *technicians_object* (section 7.15)
- *array_maint_id* - the ID of the array-based scheduled maintenance event
- *array_maint_due* – identifier of when array-based maintenance is due
- *array_maint_checker()* - identifier of when array-based maintenance is done in each year
- *onsite_vessel_id_in_use* – ID of the vessel being used for array repairs or maintenance
- *delay_status* - String identifier of the cause of delays to work, if any

7.13.1 Start

The *start* subroutine in the *array_object* class module is called during the *setup_class* key procedure via *maint_manager*, as shown in Figure 7.2 (page 44). It is used to set up the array-based aspects of the model, as well as create each of the *wec* objects. The calling procedure sends *start* the user-defined total number of WECs in the array (*no_total_wecs*), as well as the objects *vessel* and *parts* for use in the function.

The objects controlling failure and output information (*array_fail_arr*, section 7.16.1 and *array_output_arr*, section 7.17 respectively) are created and setup by calling their *start* procedures. The variable *num_wecs* is assigned to the argument *no_total_wecs* sent by *maint_manager.start*. The array is initialised by setting the *array_power* to 1 (i.e. full capacity), *state* to operating, *onsite_vessel_id_in_use* to zero (i.e. no vessel in use), and *delay_status* to "none". The value of *array_maint_due* is also initialised to equal zero, indicating that no array-based scheduled maintenance is due, unless set. This uses the assumption that OPEX calculations only start once all the WECs are installed at site in the first instance. A *for* loop then considers each of the WECs in turn, starting at 0 (*wec(0)* is only used for access to procedures) up to *num_wecs* with the identifier *i*. The *start* subroutine in each *New wec_object* is called (see section 7.14.1) with the arguments *i* (the ID of WEC), *vessel* (*vessel_object()*), *num_wecs* and *parts* (*parts_object*). The *technicians* object is also initialised here by calling its *start* procedure (section 7.15.1).

The model is currently only coded to allow one array-based scheduled maintenance event to be defined (i.e. "Moorings inspection", see section 4.1.3), although this can be modified by the user if required by following the same structure as seen throughout the *wec* object (section 7.14). After the array has been initialised and the relevant objects have been set up in *start*, the function then identified the ID of the array-based scheduled maintenance event (if there is one) defined in the *data_sheet*. To achieve this, a *for* loop considers each of the listed maintenance events using *maint_param_list.get_no_maint* (section 7.4) and identifies when the *relevance* (*get_relevance*) is for the "Array". If an array-based maintenance event is found then the variable *array_maint_id* is set to be that ID, and the counter (*count*) is updated. If the *count* value is still zero after each maintenance event in the *data_ sheet* has been considered, then *array_maint_id* is set to zero. If *count* is greater than 1 then an error message is displayed explaining that more than one array-based event has not been coded for and prompts the user to exit the program (*terminate_program*, section 7.1.8). If *count* is 1, then the staggered maintenance entry in the *data_sheet* (see section 4.1.3) must be set to "N/A" or an error message will be displayed. The variable *array_maint_checker* is re-sized to contain an Integer value for each year of the project lifetime (*no_run*). Each entry is first initialised in a *for* loop to contain 1, saying that the array-based maintenance event has already been undertaken in every year. The *for* loop is then repeated for every year (from 1 up to *no_run*) with the *Step* being taken as the user-defined *get_interval_years* entry for that category (see section 4.1.3). In each of the selected years, the relevant entry in *array_maint_checker* (i.e. *array_maint_checker(i)*, where *i* is the year identifier) is changed to contain 0, indicating that the event is yet to take place in that year.

7.13.2 Determine failure

The *determine_failure* subroutine in the *array_object* is called by *maint_manager* at every interval in order to simulate the occurrence of array-based faults and call the same procedure in every *wec* object. It is sent date information in the form *irun* (the current year) and *this_interval* (the current

interval in *irun*). Every fault category listed in the *data_sheet* (section 4.1.2) is considered in a *for* loop with the identifier *ifail*, where the total number of categories is obtained using the function *get_no_fail* in the *fail_param_list* object (section 7.3). The in-built function *With* is used to call various functions from the *fail_param_list* object without having to repeat *fail_param_list.get_fail_param(ifail)* on every line, thus making the code clearer to read.

If the relevance (*.get_relevance*) of the fault category under consideration is "Array", then the Monte Carlo analysis of simulating faults is only undertaken if the array is neither undergoing repair nor maintenance (i.e. *if state = operating*). A random number between 0 and 1 is assigned to the variable *rand* using the in-built function *Rnd*. If *rand* is greater than the probability of failure per interval for the array-based fault category (*.get_percent*) then the failure is simulated. The subroutine *add_array_fail* is used to update the *array_fail_arr* object (containing a list of current array failures) by calling *add_fail* (see section 7.16.1) with the failure category ID (*ifail = fail_cat*) as the argument. In addition, the failures output object (*fail_output_list*) is told of the failure by calling *set_total_occurrence* (see section 7.25.2) with *ifail* as the argument, thereby updating the output table of fault categories on the *results_sheet* (see section 6.1).

If the relevance (*.get_relevance*) of the fault category is "WEC", then the *determine_failure* subroutine in every *wec* object (see section 7.14.2) is called in order to run the Monte Carlo analysis of simulating failures on each device in the wave energy array. If *.get_relevance* does not match either "Array" or "WEC" then an error message is displayed and the user is prompted to exit the program (*functions.terminate_program*).

7.13.3 Determine fix

The *determine_fix* subroutine in the *array_object* is called by *maint_manager* at every interval of the year (see section 7.5.5) in order to identify times when any scheduled maintenance event is due (both array-based and WEC-based). It is sent the date information *irun* (current year) and *this_interval* (current interval in *irun*) by the calling procedure.

A *for* loop considers each of the maintenance events listed in the *data_sheet* using the function *get_no_maint* via the *maint_param_list* object (section 7.4) with the identifier *i*. The function *get_maint_interval_in_year* (see section 7.14.32) in the zero *wec* object (i.e. *wec(0)*) is used to obtain the interval in the year when the maintenance category (*i*) is due, based on the season defined in the *data_sheet* (see section 4.1.3). Although the function is located in the *wec* object, it is independent of the rest of the object's functionality. In other words, *get_maint_interval_in_year* converts a 'season' entry into an 'interval', regardless of whether that maintenance event is array-based or wec-based. The obtained interval is stored in the variable *maint_due_interval*.

If the action required to undertaken that maintenance event (i.e. *maint_param_list.get_maint_param(i).get_action_reqd*, see section 7.4) is "Retrieve WEC" then the task is WEC-based. The number of WEC-based maintenance categories (*count*) is updated before the current interval (*this_interval*) is checked against the value of *maint_due_interval*. If the current interval is greater than or equal to the 'due' interval, then a nested *for* loop considers each WEC in the project, using the identifier *k*. The procedure *set_maint_due* (see section 7.14.3) in the relevant *wec* object (i.e. *wec(k)*) is then called in order to carry out the scheduled maintenance task (*i*) on that WEC as soon as possible. However, the user-defined number of WECs allowed at the O&M base purely for maintenance (*max_wecs_offsite_maint* , see section 4.1.1) is used as an additional constraint here. The constraint is only utilised if the WEC is onsite (i.e. *if wec(k).get_state = on_site*) due to the

interaction with the cost-benefit analysis (refer to section 7.12). If the *wec* function *get_maint_due* (section 7.14.36) identifies that the maintenance task (using *count* as the argument, see section 7.14.3 for further information) is due then the function *count_wecs_offsite* is used the count the number of WECs that are not onsite (i.e. in transit or at the O&M base). If the value returned by *count_wecs_offsite* is less than *max_wecs_offsite_maint* (i.e. if there is enough space at the O&M base for the WEC) then the WEC is ready to undergo the maintenance. This is defined by sending the *wec* function *define_set_for_maint* (section 7.14.3) the String argument "Yes". Otherwise, it is sent "No", thereby restricting the WEC's maintenance until enough space is available at the O&M base.

If the *action_reqd* (see section 7.4) to undergo the maintenance task is not "Retrieve WEC", then the ID of the array-based maintenance category (*array_maint_id*, see section 7.13.1) is check against the current category under consideration (*i*). If these two values match, and if the relevant entry in *array_maint_checker* for the current year (i.e. *array_maint_checker(irun)*) is zero (i.e. array-based maintenance has not been undertaken already that year), and if current interval is beyond the *maint_due_interval*, then the value for *array_maint_due* is changed to 1, identifying that array-based maintenance is due as soon as possible. If the maintenance category is not recognised by either of the two conditions then an error message is displayed, prompting the user to end the program.

7.13.4 Attempt fix

The *attempt_fix* subroutine in the *array_object* is called by *determine_actual_fix* in *maint_manager*, as described in section 7.5.6. It is used to simulate the repairs and maintenance tasks undertaken on the array, as well as controlling the tasks carried out on each WEC. The calling procedure sends *attempt_fix* the date variables *irun* and *this_interval*, as well as the class modules *weather, vessel, parts* and *delays*. It is also sent the *ordered_list* of WEC IDs. This is either the list returned by the cost-benefit analysis (see section 7.12.5) or just the full list of WECs in ascending order. The following variables are used throughout the *attempt_fix* function:

- *i, j* - identifiers
- *fail_arr()* - used to store the list of array-based failures
- *fuel_hours* - length of the marine operation (in hours)
- *wndo_length* - length of the required weather window (in hours)
- *ops_lim_type* - type of operational limits required
- *this_tech* - technician identifier
- *delayed_by_techs* – Boolean identifier of delays due to lack of technicians
- *num_contractors_needed* – number of external contractors needed
- *perm_techs_to_assign* – number of permanent staff to assign to task
- *actions_array* - list of actions required to repair array-based faults
- *this_action* - identifier of action required
- *vessel_name_reqd* - name of vessel required for task
- *vessel_id_to_use* - ID of vessel used for current array-based task

The flowchart shown in Figure 7.6 provides visual representation of the structure, functionality and key procedures called throughout the *attempt_fix* function in *array_object*. The sources of information (i.e. the called objects and procedures) which have not been made clear in Figure 7.6 are described in the subsequent text.

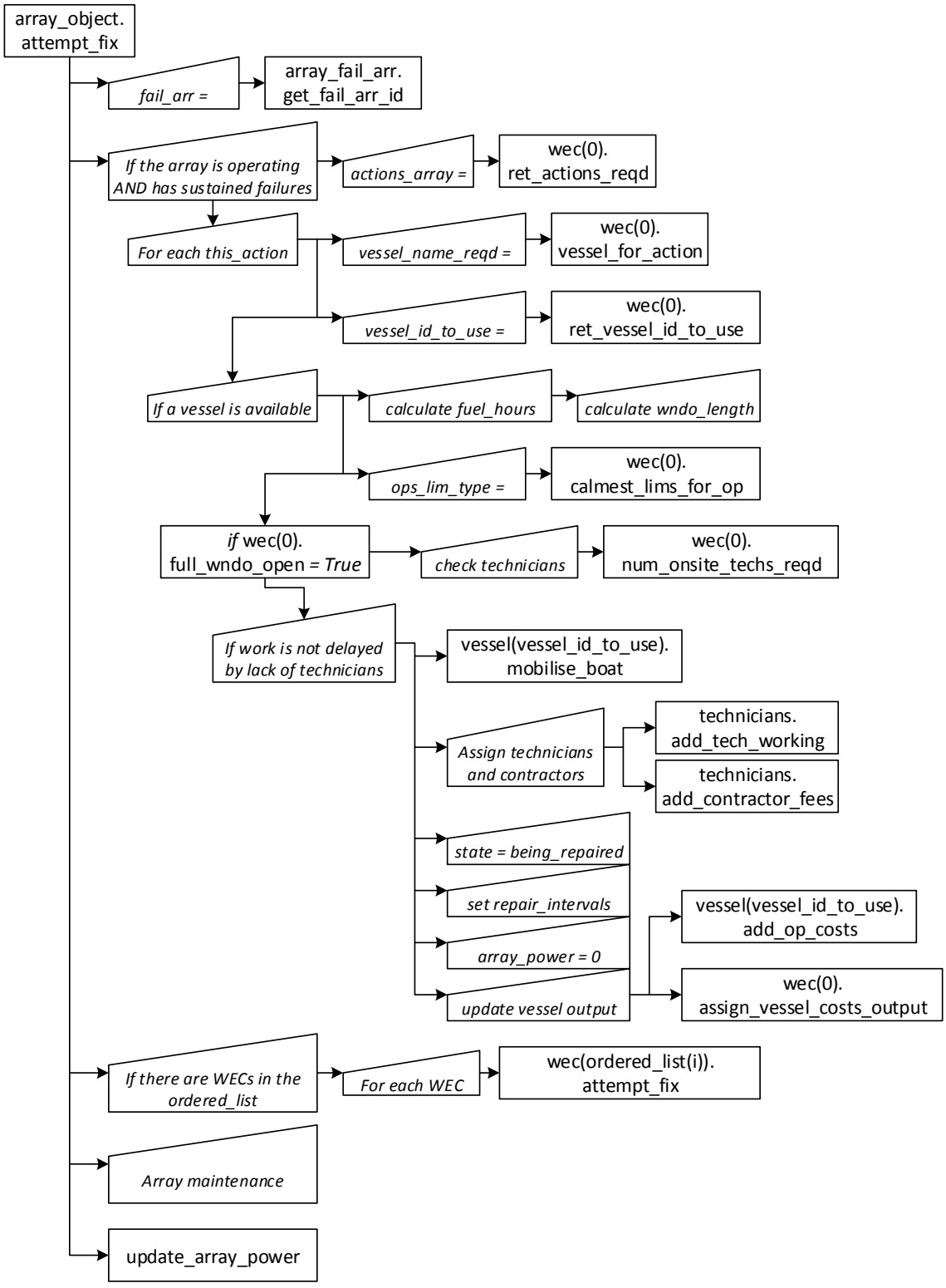

Figure 7.6. Structure of the *attempt_fix* subroutine in *array_object*

The structure of *attempt_fix* has three main sections, indicating the hierarchy of tasks:

1. Array-based failures
2. WEC-based failures and maintenance (see section 7.14.4)
3. Array-based maintenance

The list of array-based failures that have been suffered since the last repair (*fail_arr*) is obtained from the *array_fail_arr* object (see section 7.16). If no array-based failures have been sustained then the first and only entry of *fail_list* will be -5. The zero entry of the *wec* objects is only used to utilise its procedures, such as *ret_actions_reqd* (section 7.14.21), *vessel_for_action* (section 7.14.23) and *ret_vessel_id_to_use* (section 7.14.24). Although only one action is required to correct array-based failures (i.e. "moorings/subsea work"), the code loops for each *this_action* in order to follow the same format as *attempt_fix* in the *wec* object (see section 7.14.4) and provide the framework for future code modifications. An available vessel is identified if *vessel_id_to_use* is greater than zero (see section 7.14.24). The function *get_offshore_hours_subse*a (section 7.13.6) is used in conjunction with the *get_free_travel_time* value obtained from the *vessel* object (section 7.8) in order calculate the number of hours it takes to complete the marine operation (*fuel_hours*). This is rounded up to the nearest *time_step* to find the length of the required weather window in hours (*wndo_length*). The type of operational limit of weather conditions (*ops_lim_type*) is calculated using the *wec* function *calmest_lims_for_op* (see section 7.14.8). The accessibility of the weather window is assessed by calling the *wec* function *full_wndo_open* (section 7.14.11) with the relevant arguments (e.g. *wndo_length*, *ops_lim_type* etc.). The user-defined selection on whether or not external contractors can be hired to assist with maintenance is then considered (*short_term_contracts_enabled*, see sections 4.3 & 7.15.1). If contractors cannot be hired, then the work is delayed (i.e. *delayed_by_techs = True*) if the number of available technicians (*technicians.get_num_techs_avail*, section 7.15) is less that the number of technicians required for the operation (*wec(0).num_onsite_techs_reqd*, section 7.14.5).If the work is to go ahead, then the correct vessel is mobilised (*vessel(vessel_id_to_use).mobilise_boat*, section 7.8.4) and technicians are assigned to the task (*technicians.add_tech_working*, section 7.15.2). Assigning technicians also involves identifying the number of contractors required (*num_contractors_needed*), if any, and adding their fees to the output object (*technicians.add_contractor_fees*, section 7.15.3). Whilst the array is *being_repaired*, it is completely shut down to reduce risk during the offshore operation (*i.e. array_power = 0*). The number of intervals required to complete the repair/s (*repair_intervals*) is set to the number of hours (*wndo_length*) divided by the model resolution (*time_step*). The final steps in simulating the start of the array repairs include setting the vessel costs (*vessel(vessel_id_to_use).add_op_costs*, section 7.8.8) and assigning them to the failures (*wec(0).assign_vessel_costs_output*, section 7.14.12), before clarifying that the work has not been delayed (i.e. *delay_status = "none"*). If the work has been delayed by either a lack of technicians ("techs"), adverse weather conditions ("weather"), or lack of an available vessel ("vessel") then the *delays* object is updated by calling its *add_this_delay* subroutine (section 7.10.2) and setting the *delay_status* accordingly.

Following the array-based failures assessment, the code moves onto calling the *attempt_fix* subroutine in each of the *wec* objects defined in the *ordered_list* (see section 7.5.5). This is explained in more detail in section 7.14.4, but follows a somewhat similar structure to Figure 7.6.

The subroutine then moves onto assessing array-based maintenance, with the category ID (*array_maint_id*) identified during the *start* procedure (section 7.13.1). If a maintenance event has been defined for the array (i.e. *if array_maint_id > 0*), then it is simulated in a very similar way to array-based failures, as shown in Figure 7.6 and described in the previous paragraphs. The key difference is that a list of the *actions_reqd* does not need to be obtained because aspects of the task, such as *get_vessel_reqd* and *get_hours_offshore*, can be identified directly from the maintenance parameter object (i.e. *maint_param_list.get_maint_param(array_maint_id)*, section 7.4). The task is only carried out if *array_maint_due*, defined during *determine_fix* (section 7.13.3), is set to 1.

The array power is then updated using the subroutine *update_array_power* (see section 7.13.5).

7.13.5 Update array power

The subroutine *update_array_power* is called by two procedures in *array_object* – *attempt_fix* (section 7.13.4) and *next_interval* (section 7.13.7) – in order to update the value of *array_power*. This procedure is required due to the fast that the model can involve array-based failures, maintenance and marine operations, as well as WEC-based aspects. In order words, the total power output of the entire array (i.e. the revenue-generating power) will not necessarily be equal to the sum of the power output of all the WECs in the project if there are array-based aspects involved.

In *update_array_power*, the *array_power* is set to zero if any array-based repairs or maintenance tasks are taking place (i.e. *if state = being_repaired*). Otherwise, the *array_power* is initialised to equal the sum of the WECs power using the function *sum_wecs_power* (see next paragraph). The list of array-based failures is then stored in the variable *fail_arr* by using the array failure object (array_fail_arr, section 7.16.1). If there are any failures, then each one is considered in a *for* loop and the power loss is subtracted from *array_power* at each step. If the value of *array_power* is negative at the end of this loop, then it is set to zero (i.e. can never have negative power output).

The function *sum_wecs_power* is used to calculate the total power output from all the WECs in the wave energy array. The return value, *temp_sum*, is initialised to zero. Each *wec* object is then considered in a *for* loop with the identifier *i*. The power output in the *wec* object is a value between 0 and 1, relating the capacity of that WEC. Therefore, this value (obtained using *wec(i).get_wec_power*) must be divided by the number of WECs in the array (*num_wecs*) in order to convert it in array-based power (between 0 and 1). Each converted *wec_power* is added to the *temp_sum* throughout the *for* loop. The use of the Double data type can cause problems for the VBA code where there are very small rounding errors. This is remedied by assigning *temp_sum* the value of 1 or 0 if the obtained value is within 10×10^{-10} of that figure. The function name, *sum_wecs_power*, is set to be *temp_sum* in order to be recognised by the calling function *update_array_power*.

7.13.6 Hours offshore for subsea work

The function *get_offshore_hours_subsea* is called by the *attempt_fix* subroutine (also in *array_object*, section 7.13.4) in order to update the number of hours is takes to completely repair an array-based failure/s. It is sent the list of array-based failures currently sustained (*fail_arr*), the number of transit hours required by the selected vessel (*transit_hours*), and the *weather* object.

Firstly, the return value *temp_hours* is initialised to be zero. This function is only called if array-based failures have been sustained, but the condition is checked (i.e. *if fail_arr(1) > 0*) anyway for completion. A *for* loop considers each entry in the *fail_list* with the identifier *i*. For the first failure, *temp_hours* is set to be the value of the *get_hours_offshore* function obtained from the failure parameters object (*fail_param_list.get_fail_param(fail_arr(i))*). If multiple failures have occurred, then it is assumed that each one will not take more than three hours, in addition to the first repair. This assumption is based on the idea that multiple failures can be repaired in parallel, or at least planned so that total time spent offshore is minimised. After each failure has been considered, *temp_hours* is updated to include the *transit_time*, thereby becoming the total number of hours required for the full marine operation. If marine operations are constrained to daylight hours (i.e. *if night_ops_on = False*) then it is possible that certain array-based failures could take longer to repair

than is possible given these restrictions. This aspect should be considered when defining the failure categories (section 4.1.2) but is incorporated into this function by modifying *temp_hours* if it is greater than the longest period of daylight throughout the year for the specified site. This period is identified by using the *weather* function *longest_daylight_wndo* (see section 7.6.5). The check is undertaken by converting the *temp_hours* value into intervals (i.e. dividing by *time_step*). The *get_offshore_hours_subsea* is set to be *temp_hours* to be returned to *attempt_fix*.

7.13.7 Next interval

The subroutine *next_interval* is called by the procedure of the same name in the *maint_manager* class module, as described in section 7.5.8. It is used to set the array up for the next interval in the simulation, as well as calling the *next_interval* subroutine for each *wec* object (section 7.14.14). It is sent the date information by *maint_manager* in the form *irun* (current year) and *this_interval* (current interval in *irun*). It is also sent the names of the relevant objects *weather, vessel, revenue, parts and delays* so their procedures can be accessed, as well as the total number of WECs in the array (*num_wecs*).

Initially, the lost revenue caused by any failures or maintenance task throughout the array (including on WECs) is assigned for producing the outputs tables seen in the *results_sheet* (see section 6.1). This is achieved by calling the function *assign_lost_revenue_fails_maint*, which is described in detail in section 7.13.8.

If the array is undergoing repairs or maintenance (i.e. *if state = being_repaired*) then the simulation is set up for the next interval by subtracting 1 from the number of intervals remaining to complete the task (*repair_intervals*)

Permanently employed technicians who have completed their current repairs or maintenance tasks are reset by calling the *next_interval* subroutine in the *technicians* object (see section 7.15.4). The subroutine *next_interval* in each of the *vessel* objects (section 7.8.4) is also called to update the number of intervals each vessel has remaining on a marine operation.

The list of array-based failures is stored in the variable *fail_arr* using the *array_fail_arr* object (section 7.16.1). If the array is *being_repaired* and if there are no more intervals remaining on the task (i.e. *if repair_intervals <= 0*) then the output information is updated. For array failures, this involves checking each action required (*this_action*) and demobilising the vessel used for the operation (*onsite_vessel_id_in_use*, set by *attempt_fix*) by calling *vessel.demobilise_boat* (see section 7.8.5). The output information for the array is updated by calling the *fail_costs* procedure in the *array_output_arr* object (section 7.17.2), whilst the table of failure categories is updated with *set_costs_repair* in *fail_output_list* (section 7.25.2). The object controlling current array-based failures, *array_fail_arr*, is reset by calling its *start* subroutine (see section 7.16.1). A similar process is undertaken when array-based maintenance is the task that has just been completed. This is identified when the *state* is *being_repaired* but there are no array-based failures (i.e. *fail_arr(1)* is not greater than 0). The correct vessel is demobilised in the same way as the failures section. The output information is updated by calling the subroutines *add_maint_ costs* (section 7.17.2) and *set_costs_maint* (section 7.26.2) in the objects *array_output_arr* and *maint_output_list* respectively. In addition, the relevant entry in the array maintenance checker (i.e. *array_maint_ checker(irun)*) is set to 1, saying that maintenance has been completed in that year, whilst the value of *array_maint_due* is reset to zero. Once the output information has been updated following a successful marine operation, the array *state* is reset to *operating*.

The subroutine *next_interval* is then called for each *wec* object in order to set the WECs up for the next interval and assigned output information if required. This is described in greater detail in section 7.14.14.

The array power is updated for completion by calling the procedure *update_array_power*, described in section 7.13.5. This leads on to the availability of the array being updated for the model outputs by calling *avail_add* in the *array_output_arr* object (section 7.17.3). The revenue generated by the array at that interval is also updated for the outputs by calling *update_rev* in the *revenue* class module (section 7.7.4).

7.13.8 Assign lost revenue for failures and maintenance

The subroutine *assign_lost_revenue_fails_maint* is called at the beginning of the *next_interval* procedure, also in the *array_object* (section 7.13.7), in order to assign shares of the total lost revenue at each interval to the relevant failure categories and maintenance tasks. It is sent the current year (*irun*) and current interval (*this_interval*), as well as the *revenue* object. Assign lost revenue means that the costs of improving certain components or O&M strategies can be justified. In other words, the user can identify how much money they should spend on improving a particular component in order to see economics benefits in future wave energy arrays. This is much more applicable to the components on each WEC, rather than array-based failures. Therefore, the subroutine gives WEC-based failure categories priority when assigning lost revenue. This hierarchy of assigning lost revenue is contained within the structure of *assign_lost_revenue_fails_maint*, as shown in Figure 7.7.

Revenue is only lost if the array is not operating at full capacity (i.e. *if array_power < 1*). Initially, the total power loss incurred from all the WECs in the array is calculated by looping through each *wec* object and adding the WEC power loss (converted to array power loss, i.e. *(1 - wec(i).get_wec_power) / num_wecs*) to the Double variable *sum_wecs_power_loss*. As with the *sum_wecs_power* function (see section 7.13.5), the final value of *sum_wecs_power_loss* must be adjusted for rounded errors. The total share of lost revenue assigned to the WECs (*wecs_share*) is set to 1 if *sum_wecs_power_loss* is equal to (or greater than) the array power loss (i.e. *1 - array_power*). Otherwise, the *wecs_share* is set to be the portion that *sum_wecs_power_loss* has on the array power loss, with *array_share* accounting for the remaining lost revenue (i.e. *1 - wecs_share*). Four key variables are initialised to zero before the subroutine's main processes are undertaken; *fails_count* (the number of fault categories to assign), *maint_count* (the number of scheduled maintenance events to assign), *fails_share* (portion to be assigned to the failures), and *maint_share* (portion to be assigned to the maintenance events).

The subroutine's purpose is achieved by checking every failure and maintenance category that has had any impact on overall array power loss and assigning each one its appropriate share of the associated lost revenue. This information is stored in VBA array variables named *fails_store_array* and *maint_store_array*. Each one is a two-dimensional array storing three pieces of information about each task:

1. The ID of the category - e.g. *fail_arr(ifail)*
2. The share of lost revenue - e.g. calculated with *calc_fail_share* (section 7.13.11)
3. The state and delay status - e.g. "onsite" & *wec(i).get_delay_status* (section 7.14.36)

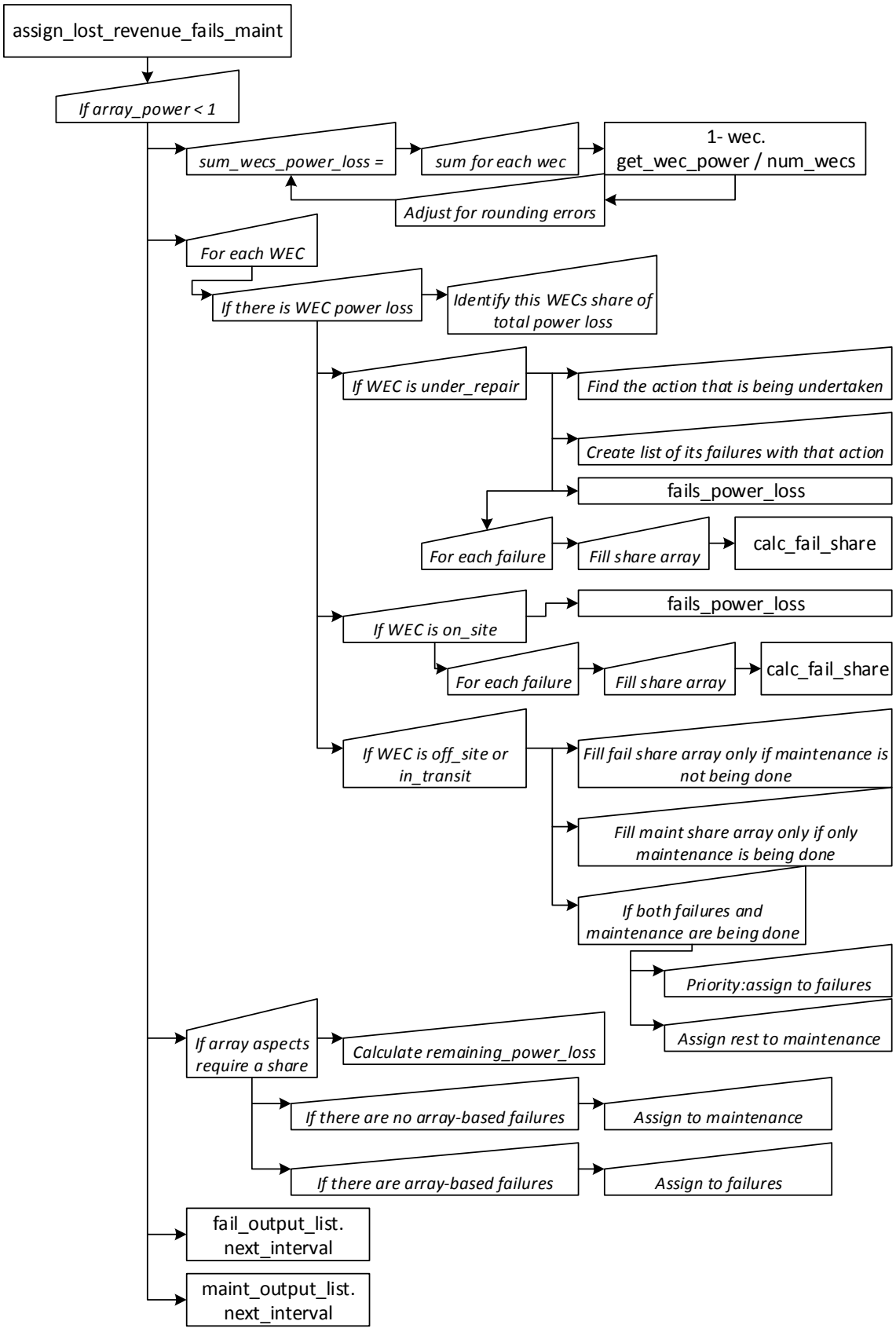

Figure 7.7. Structure of *assign_lost_revenue_fails_maint*

As shown in Figure 7.7, *assign_lost_revenue_fails_maint* then enters a *for* loop where each WEC in the array is considered (using the identifier *i*). The WEC's failures and/or maintenance categories are only assigned lost revenue if there is any loss of *wec_power* (i.e. *if wec(i).get_wec_power < 1*). The portion of total power loss from all WECs attributed to the WEC under consideration is stored in the variable *this_wecs_portion*. The list of failures sustained by the WEC (*fail_arr*) is obtained from the failures object related to the *wec* class module (i.e. *wec(i).get_fail_list.get_fail_arr_id*).

If the WEC is undergoing on site repairs (i.e. *if wec(i).get_state = under_repair*) then the action that has taken priority must be identified. This assumes that only one set of onsite repairs (e.g. replacing a PTO unit) can be done at a time, as described in greater detail in the *wec* subroutine *attempt_fix* (see section 7.14.4). The current action (*this_action*) is identified using the *wec* functions *ret_actions_reqd* and *ret_action_onsite_priority* (see sections 7.14.21 and 7.14.22 respectively). The list of failures (*fail_arr*) is then modified (and renamed as *new_fail_arr*) in order to identify all the failures requiring *this_action* using the *wec* function *ret_action_fails* (section 7.14.25). The power loss incurred by these failures is stored in the variable *power_loss_from_ these_fails* using the function *fails_power_loss* (section 7.13.9). Each of the fault categories listed in *new_fail_arr* are considered in turn in a *for* loop using the identifier *ifail*. In order to re-size the *fails_store_array*, the value of *fails_count* is added to for each category. As stated previously, the *fails_store_array* is a 2D array with three entries for each fault category. The first entry (i.e. *fails_ store_array(1, fails_count)*) stores the ID of the category (i.e*. new_fail_arr(ifail)*). The second entry (i.e. *fails_ store_array(2, fails_count)*) stores the share of overall lost revenue to be assigned to that category, obtained by the *calc_fail_share* function (section 7.13.11). The third and final entry (i.e. *fails_ store_array(3, fails_count)*) stores String information about the status of the failure ("onsite repair" in this case).

If the WEC is on site but not undergoing a repair (i.e. *ElseIf wec(i).get_state = on_site*) then the power loss from the failures (*fail_arr*) is obtained from *fails_power_loss* (section 7.13.9). Then a *for* loop considers each failure in turn and fills the *fails_store_array* appropriately, as described above. The third entry of the array (i.e. *fails_store_array(3, fails_count)*) stores the *delay_status*, as well as the String information "onsite".

If the WEC is either offsite or in transit, however, then the lost revenue might be due to maintenance tasks as well as failures. The scenario where the lost revenue is solely due to failures is identified using the condition *if wec(i).any_maint_due = False* (see section 7.14.34). The WEC is only in this *state* due to failure categories that need retrieval. Therefore, the function fails_power_ loss_retrieve (section 7.13.10) is used to calculate the value of *power_loss_from_these_fails*. The *for* loop adds information about each failure to *fails_store_array* as before, but only if the action for that failure (i.e. fail_param_list.get_fail_param(fail_arr(ifail)).get_action_reqd) is "Retrieve WEC". The *calc_fail_share* function (section 7.13.11) is again used to assign the failure for each failure, with the appropriate parameters as arguments (such as wec(i).num_retrieval_fails, section 7.14.33). If the status of the WEC is "in transit" then the work is not delayed. However, if the WEC is "offsite" then installation could be delayed, meaning the third entry of *fails_store_array* should also contain the *delay_status*. If the WEC is undergoing maintenance only (i.e. no failures and *wec(i).any_maint_ready = True*) then each scheduled maintenance category requiring the action "Retrieve WEC" is considered in a *for* loop. The *maint_store_array* is filled in the same way as *fails_store_array*, with the key difference being that the function *calc_fail_share* is not used to fill the second entry. Instead, the share of lost revenue attributed to that maintenance is event is calculated by the equation:

$$maintenance\ (imaint)\ share = \frac{this\_wecs\_portion}{number\ of\ maintenance\ events\ due} \times wecs\_share$$

Where the number of maintenance events due is obtained by the *wec* function *get_num_wec_maints_due* (section 7.14.34). If the state of the WEC (*off_site, being_removed* or *being_installed*) is not due solely to either failures or maintenance, then it is a combination of both. In this case, each one must be assigned a share of the total power loss (i.e. full power loss because the WEC is offsite). These shares are stored in the variables *fails_share* and *maint_share*. The *fails_share* is assigned based on the power loss caused by those failures (*power_loss_from_these_fails*), whilst the *maint_share* is calculated based on the fact that maintenance causes full power loss (i.e. the WEC to go offsite).

Note: this assumption means that no scheduled maintenance is carried out whilst the WEC in offshore – a justified assumption based on WEC developer experience.

The share arrays (*fails_store_array* and *maint_store_array*) are filled in a similar way as described previously, with the only difference being that the assigned share (i.e. the second entry) incorporates the relevant multiplier (*fails_share* and *maint_share* respectively). This completes the WEC-section of the subroutine *assign_lost_revenue_fails_maint*.

If the share of lost revenue attributed to the array (*array_share*) was not identified as zero then the *remaining_power_loss* to be assigned is calculated (i.e. *(1 - array_power) - sum_wecs_power_loss*). The variable *fail_arr* is reset to contain to list of array-based failures. If user has defined an array-based scheduled maintenance event (i.e. subsea moorings inspection, see section 7.13.4) and it is being undertaken, then the *maint_store_array* is updated to contain the relevant information. The full *array_share* of lost revenue is assigned to the event. However, if array-based failures have occurred then the *fails_store_array* is updated in the same manner as described previously, using the functions *fails_power_loss* (section 7.13.9) and *calc_fail_share* (section 7.13.11).

Following the described processes, the relevant output objects are updated. This only occurs if there have been failures and/or maintenance categories stored in the variables *fails_store_array* and *maint_store_array*, identified if *fails_count* and/or *maint_count* respectively are greater than zero. In each case, the outputs objects *fail_output_list* and *maint_output_list* keep track of the lost revenue details with their *next_interval* functions (see sections 7.25.3 and 7.26.3 respectively)

7.13.9 Power loss from failures

The *fails_power_loss* function is utilised by the *assign_lost_revenue_fails_maint* subroutine in the *array_object* class module, as described in section 7.13.7. It takes a list of failures (*fail_arr*) and calculates the sum of the power loss on the entire array caused by each one. To achieve this, the function obtains the power loss for each failures defined on the *data_sheet* (section 4.1.2) using *get_power* in the relevant object (i.e. *fail_param_list.get_fail_param(fail_arr(i)).get_power*, section 7.3).

7.13.10 Power loss from failures that need retrieval

The *fails_power_loss_retrieve* function is also utilised by the *assign_lost_revenue_fails_maint* subroutine in the *array_object* class module (section 7.13.7). It operates in exactly the same way as *fails_power_loss* (section 7.13.10) but only adds the power loss from a failure to the return value if

the action required for that failure is “Retrieve WEC” (i.e. *fail_param_list.get_fail_param(fail_arr(i)).get_action_reqd*, section 7.3

7.13.11 Calculate failures share

The *calc_fail_share* Double function is utilised by the *assign_lost_revenue_fails_maint* subroutine in the *array_object* class module, as described in section 7.13.7. It is used to calculate the share of lost revenue to be assigned to a particular failure. To achieve this, it is sent the following arguments:

- *num_fails* - the total number of failures to be assigned in this section
- *this_fail* - ID of this failure
- *total_power_loss* - power loss from all failures in this section
- *system_portion* - the portion to be assigned to this system
- *this_share* - the share of all WECs (*wecs_share*) or the array (*array_share*)

If the failures in the particular section of *assign_lost_revenue_fails_maint* causes any power loss on the array (i.e. *if total_power_loss > 0*) then the portion of *total_power_loss* assigned to *this_fail* is identified and stored in the variable *this_fail_portion*. The share of the overall lost revenue from the array at that interval is then calculated by multiplying *this_fail_portion* by the *system_portion* and *this_share*. This method means that if a reduced-capacity WEC has suffered two failures, but one has had no effect on power, then only the other failure is assigned any of the blame. If none of the failures have any effect on array power loss, then the blame is split evenly between them.

7.13.12 Print interval

The subroutine *print_interval* in the *array_object* class module is only called when a ‘full run’ process is taking place in order to print array-relevant information to the ‘run sheets’ (see section 6.2). It is called by the *maint_manager* object, as shown in Figure 7.3 (page 52), with arguments including date information (i.e. *irun* and *this_interval*) and the starting column of particular sections of *run_sh* (e.g. *wec_start_col*).

The subroutine starts by printing the array-based headers if it is the first interval of a year (i.e. *if this_interval = 1*). As described in section 6.2, this includes “Array failures” and “Array power capacity”. In the “Array failures” column, the *state* of the array (e.g. “Operating”) is printed along with a list of the array-based failures currently sustained. The maximum severity of the failures is obtained using the *get_fail_number* function in the *array_fail_*arr object (see section 7.25) and the cell is filled with the appropriate colour (i.e. red for major failures, amber for intermediate, and green for minor). If the array-based maintenance category (“Moorings inspection”) is taking place then the cell reads "Subsea inspection" and is filled dark red. The cells in the “Array power capacity” column contain the *array_power* (see section 7.13.5).

The value of *wec_start_col* is updated for every *wec* object before the *print_interval* subroutine is called, thereby printing the information for each WEC in the array (see section 7.14.20). The value of the argument *techs_start_col* is then sent to the *print_interval* subroutine in the *technicians* object (section 7.15.5), thereby printing information about the technicians at the O&M base to the *run_sh*, as well as information about external contractors.

7.13.13 Post process

The *post_process* function in the *array_object* is called by *maint_manager* (section 7.5.9) in order to control the printing of output information to the 'Results' spreadsheet (section 6.1). To achieve this, the function prints output data about each WEC using the *draw* procedure in the relevant output objects (i.e. *wec(i).get_wec_output_list .draw*, section 7.18.4). The WEC output list objects are stored in the variable *wec_output_arr_loc* so that the procedure *draw_all_wecs* in the array output object (*array_output_arr*, section 7.17.7) can print the relevant output data. Also, the subroutine *draw* in the *array_output_arr* object (section 7.17.4) is called to print the output data pertaining to the entire wave energy array. *Post_process* is a function because it returns the *array_output_arr* object so that it can be used by *maint_manager*.

7.13.14 Get functions

Other class modules can obtain information contained within the *array_object* by using the following functions:

- *get_num_wecs* - the number of WECs in the array, *num_wecs*
- *get_technicians_object* - the *technicians* object
- *get_wec* - the *wec_object* for a specified WEC (*this_wec*)

7.14 WEC OBJECT

The *wec_object* class module is referred to simple as *wec* by the *array_object*. A new *wec* object is created for every WEC in the wave energy array, as defined by the user on the 'Inputs' spreadsheet (see section 4.1.1), during the *setup_class* procedure (see Figure 7.2, page 44). Each *wec* object is used to simulate the repairs and maintenance events undertaken on the device. As discussed in section 7.13.1, the *start* subroutine in *array_object* also creates an extra *wec_object* with the zero entry (i.e. *wec(0)*) in order to access procedures stored in the class module. The *wec_object* class module is also utilised during the cost-benefit analysis part of the O&M model, as described throughout section 7.12. A large number of variables are used throughout *wec*, shown in Table 7.1.

Table 7.1. Variables used throughout the *wec_object* class module

Variable name	Data type	Description
state	*wec_state*	Current state of this WEC
wec_id	Integer	ID of this WEC
wec_power	Double	Power capacity of this WEC (0 to 1)
wec_fail_arr	Object *wec_fail_list*	WEC failures object
wec_output_arr	Object *wec_output_list*	WEC output object
intervals_off_site	Integer	Number of intervals the WEC has left offsite
retrieval_ints	Integer	Number of intervals the WEC has left to complete retrieval
install_wndo_type	Integer	Type of operational limits required for WEC installation
install_hours	Double	Number of hours required for WEC installation
install_num_techs	Integer	Number of technicians required for install

install_remaining_ints	Integer	Number of intervals the WEC has left to complete installation
install_vessel_name	String	Name of vessel required for WEC installation
install_vessel_id_in_use	Integer	ID of vessel used for marine operation for this WEC
wec_maint_cat	Integer()	List of IDs of WEC-based maintenance events
maint_due	Integer()	Identifier of when each WEC-based maintenance event is due
maint_checker	Integer()	Identifier of if each WEC-based maintenance event has been done in each year
set_for_maint	*yes_no*	Identifier of if WEC-based maintenance events are constrained by O&M base space
delay_status	String	Cause of delay to work for this WEC, if any
offsite_failures_array	Integer()	List of IDs of faults sustained by the WEC requiring retrieval
offsite_fails_ints_worked	Integer()	Number of intervals worked on each repair of retrieval faults
offsite_maint_array	Integer()	List of IDs of maintenance events being undertaken
offsite_maint_ints_worked	Integer()	Number of intervals worked on each maintenance event
offshore_repair_time	Integer	Number of intervals remaining to complete an onsite (i.e. offshore) repair of a fault
onsite_vessel_id_in_use	Integer	ID of vessel used for onsite repair task
delayed_by_techs	Boolean	If work is delayed by a lack of O&M base technicians
num_contractors_needed	Integer	Number of external contractors required for task
perm_techs_to_assign	Integer	Number of O&M base technicians to assign to task
replacement_parts_delayed	Boolean	Identifier of if tasks are waiting for replacement parts to be delivered

Note that in the 'Data type' column in Table 7.1, a VBA array is identified by an in-built data type followed by parentheses (e.g. Integer()). Custom data types (see section 7.1.1) are identified by italic font.

7.14.1 Start

The *start* subroutine of each *wec* is called by *array_object* during the *setup_class* procedure, as shown in Figure 7.2 (page 44). It is sent the arguments of *i* (the position in the *for* loop, see section 7.13.1), *vessel* (to use the procedures in the *vessel_object* class modules), *num_wecs* (the total number of WECs in the array), and *parts* (the *parts_object*). The *start* subroutine is used to initialise the variables for each *wec* in the array and read relevant information from the *data_sheet*.

Firstly, the *wec_id* is set to be the argument *i* sent by the calling procedure. The remainder of the subroutine is only utilised for WECs in the array (i.e. *if wec_id > 0*) because the zero *wec* entry (i.e. *wec(0)*) is used solely for access to certain procedures.

The objects controlling the failures sustained by the WEC (*wec_fail_arr*) and the output information (*wec_output_arr*) are created an initialised by calling their *start* procedures (*wec_fail_arr.start* and *wec_output_arr.start*, sections 7.16.3 and 7.18.1 respectively). The *state* is set to *onsite* and the *wec_power* is set to 1, saying that the model simulations start at the point when all WECs have been fully commissioned and deployed for the first time.

The user-defined entries related to WEC installation listed in the *data_sheet* (section 4.1.1) are stored in the relevant objects; *install_wndo_type, install_vessel_name, install_hours and install_num_techs* (described in Table 7.1). The other two installation-related variables, *install_remaining_ints* and *install_vessel_id_in_use*, are initialised to zero, saying that no install operation is currently taking place.

Each WEC-based scheduled maintenance event listed in the *data_sheet* (see section 4.1.3) is then considered using a *for* loop (with the identifier *k*) and an *if* condition identifying the events that have the *action_reqd* of "Retrieve WEC" (via *maint_param_list*, section 7.4). The variable *count* is used to keep track of how many such events have been identified. For each one, the variable array *wec_maint_cat* is re-sized from 1 to *count* and the new entry (i.e. *wec_maint_cat(count)*) is set to contain the ID of the maintenance event (*k*). The array *maint_due* is resized in the same way, with the new entry (i.e. maint_due(count)) set to zero, thereby saying that the corresponding maintenance category is not due yet.

Note: the position of each WEC-based maintenance event in the arrays *wec_maint_cat, maint_due* and *maint_checker* does not necessarily correspond to the ID of the event, unless they are listed together in the 'Inputs' spreadsheet.

The variable *maint_checker* is re-sized as a two-dimensional array, with the first dimension containing entries for each year (i.e. 1 to *no_run*) and the second dimension corresponding to the maintenance event under consideration (i.e. 1 to *count*). It should be noted that the entries need to be this way round because VBA only allows the last dimension of a multi-dimensional array to be expanded. For each year in the project lifetime (i.e. *yr_count* from 1 to *no_run*), the relevant entry in *maint_checker* (i.e. *maint_checker(yr_count, count)*) is initialised to 1, saying that the maintenance event has already been done in that year. The user-defined information about the maintenance interval (i.e. *get_interval_yrs*) is then obtained from the *maint_param_list* object and stored in the Integer variable *my_step*. If the user has selected no staggered maintenance for that category (i.e. *get_staggered_maint = "No"*, see section 7.4), then all the WECs will undergo the maintenance event *k* (also identified by *wec_maint_cat(count)*) in the same year. This condition is met by looping through all years in the project lifetime (i.e. *yr_count* from 1 to *no_run*) with

my_step as the *Step*, and setting to relevant entry in *maint_checker* (i.e. *maint_checker(yr_count, count)*) to zero, thereby saying that the maintenance event has not been done in those years yet. If *my_step* is equal to half the project lifetime, then only that year (*my_step*) is changed to zero for that maintenance event (i.e. *maint_checker(my_step, count) = 0*). This condition avoids the "Major components refit" maintenance task being undertaken in the final year of the project lifetime (e.g. year 20 if *my_step* is 10). If the user has chosen the maintenance to be staggered (i.e. *get_staggered_maint = "Yes"*, see section 7.4), then the start year of the maintenance must first be identified. This is assigned by finding the portion of WECs to start undergoing the maintenance event in the years 1 to *my_step* using a *for* loop and the identifier *j*. In each case, if the *wec_id* is less than or equal to that portion (i.e. *num_wecs * (j / my_step)*), then the *start_yr* is identified as the year under consideration (*j*). A new *for* loop then considered each year in the project lifetime (i.e. up to *no_run*) from *start_yr* with the *Step* of *my_step*, filling the relevant entry in *maint_checker* (i.e. *maint_checker(yr_count, count)*) with zero. Error handling is in place if the *start_yr* value is not identified correctly, or if the user-defined entry of staggered maintenance is invalid.

Following the maintenance events loop, the remaining variables are initialised; *replacement_parts_delayed* to False, *delay_status* to "none", and *onsite_vessel_id_in_use* to zero.

7.14.2 Determine failure

The subroutine *determine_failure* is used to simulate the occurrence of faults on WECs using a Monte Carlo analysis. It is called by the procedure of the same name in *array_object* (see section 7.13.2) with the arguments *fail_cat* (the ID of the fault category) and *num_wecs* (the number of WECs in the array).

The analysis is only undertaken if the WEC *state* is *onsite*. A random number between 0 and 1 is generated using the *Rnd* function and stored in the Double variable *rand*. If this value is greater than the probability of the *fail_cat* not failing in a given interval (identified in *fail_param_list* with *get_percent*, section 7.3) then the failure is simulated. The function *add_wec_fail* is called with *fail_cat* as the argument in order to call the *add_fail* subroutine in the failures object *wec_fail_arr* (see section 7.16.3). In addition, the *set_total_occurrence* subroutine in the output object for the fault categories (printed on the 'Results' spreadsheet, see section 6.1), *fail_output_list*, is also called to update the total occurrence of that failure. The *wec_power* is then updated by subtracting the power loss sustained due to that failure from the current *wec_power*. It is important to note that the power loss listed in the relevant *fail_param_list* object (i.e. *get_fail_param(fail_cat)*) is in the format of array-based power loss. Therefore, it must be converted into wec-based power loss by multiplying it (i.e. *get_power*) by the total number of WECs in the array (*num_wecs*) before being subtracted from the current *wec_power*. Finally, *wec_power* is set to be zero if the sustained failure would otherwise make it a negative value.

7.14.3 Set for scheduled maintenance

The subroutine *set_maint_due* is called during the *determine_fix* procedure in the *array_object* class module (section 7.13.3) in order to determine if a given WEC-based scheduled maintenance event is to be undertaken. It is sent the date information *iyear* (the current year) and *this_interval* (the current interval in *iyear*) as arguments, as well as the identifier *this_entry*. The value of *this_entry* corresponds to the current number of WEC-based maintenance events considered in the *for* loop in *array_object.determine_fix* where *set_maint_due*. This means that *this_entry*

corresponds to the relevant entry in the maintenance arrays in the *wec* object (i.e. *wec_maint_cat(this_entry), maint_due(this_entry) and maint_checker(iyear,this_entry)*).

The subroutine *set_maint_due* only continues if the WEC *state* is *onsite* and if the maintenance category under consideration has not already been done and was assigned to be undertaken in this year (i.e. *if maint_checker(iyear, this_entry) = 0*). The procedure *array_object.determine_fix* already identified that *this_interval* is greater than the interval specified for maintenance (*maint_due_interval*, section 7.13.3). Therefore, if the conditions have been met then the relevant entry in the *maint_due* array (i.e. *maint_due(this_entry)*) is set to 1, thereby identifying that maintenance is due. It is possible that other challenges, such as adverse weather conditions, will cause some maintenance events to face significant delays. In rare cases, this could cause the event to carry over into the next year. This scenario is accounted for by changing the relevant entry in the *maint_checker* array (i.e. *maint_checker(iyear + 1, this_entry)*) to zero if *this_interval* is the last interval of the year (i.e. equal to *no_intervals*) and *iyear* isn't the final year of the project lifetime (i.e. is less than *no_run*).

The subroutine *define_set_for_maint* is also called by the *determine_fix* procedure in *array_object* in order to utilise the user-defined constraint on the number of WECs allowed at the O&M base solely for maintenance (see section 4.1.1). If there is enough space at the O&M base, then *define_set_for_maint* is sent the *definition* String value *"yes"*, which then makes the *set_for_maint* variable equal to *yes* (using the *yes_no* custom data type, section 7.1.1). Otherwise, *set_for_maint* is set to *no*, thereby constraining the WEC-based maintenance events.

7.14.4 Attempt fix

The subroutine *attempt_fix* is called by the procedure of the same name in the *array_object* class module, as described in section 7.13.4, in order to simulate the start of marine operations to repair failures or undertake scheduled maintenance on the WEC represented by the *wec* object. To achieve this, it is sent the date information of *irun* (current year) and *this_interval* (current interval in *irun*), as well as the class modules it needs to access; *weather, vessel, parts* and *delays*. The subroutine is also sent the total number of WECs in the array (*num_wecs*) and the number of WECs that are not on site at the moment (*num_wecs_offsite*). A number of variables are used throughout *attempt_fix*:

- *j* - identifier
- *fail_arr* - stores a list of WEC-based failures
- *fuel_hours* - actual number of hours to undertake a marine operation
- *wndo_length* - number of hours to undertake marine operation, rounded up to the nearest full *time_step*
- *ops_lim_type* - type of operational limits for marine operation
- *vessel_name_reqd* - name of vessel required for marine operation
- *vessel_id_to_use* - ID of vessel to use for marine operation
- *this_tech* - identifier of technicians
- *actions_array()* - list of actions required to correct faults
- *this_action* - name of an action required to correct faults
- *new_fail_arr()* - stores a modified list of WEC-based failures
- *this_part_type* - name of spare part required for correcting faults whilst offshore

The main structure and functionality of the subroutine *attempt_fix* is shown in Figure 7.8.

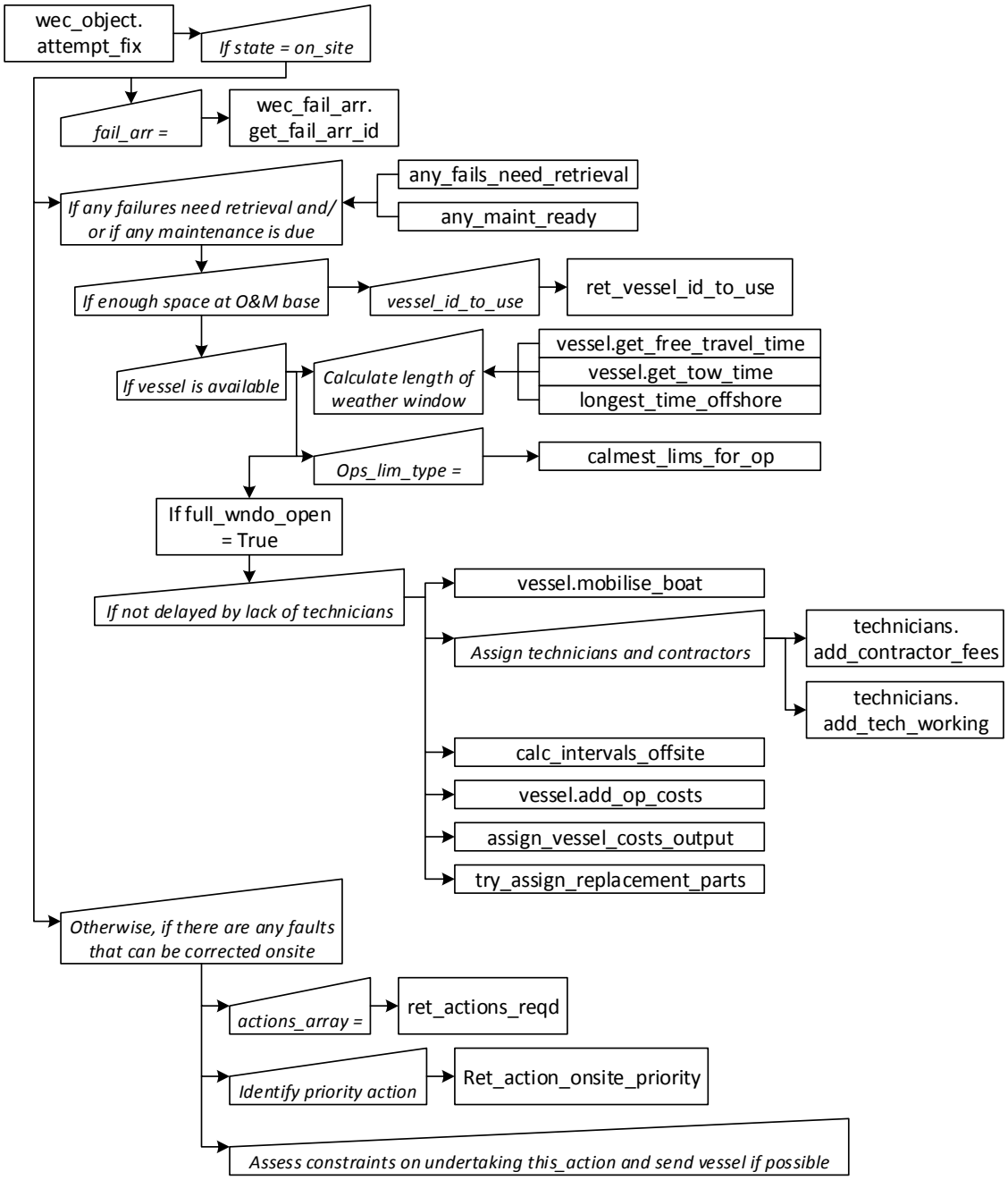

Figure 7.8. Structure of *attempt_fix* in *wec_object*

As indicated by Figure 7.8, *attempt_fix* is structured in a way that the priority is given to actions which require the WEC to be retrieved and taken to the onshore or quayside O&M base. This includes failures which require retrieval, as well as any WEC-based maintenance event. The second level of the hierarchy is for failures which can be corrected whilst the WEC in onsite by replacing certain parts (i.e. a PTO unit or instrumentation box, see section 7.9). In the case where a WEC has sustained two failures, one which requires retrieval and one which needs spare parts, then both faults are corrected once the WEC is at the O&M base. This hierarchy is made clear throughout this section and with the descriptions of the procedures used in the *wec* object.

The subroutine only continues if the *state* of the WEC is *onsite*. A list of the failures sustained by the WEC is obtained using the *get_fail_arr_id* function in the failures object *wec_fail_arr* (section 7.16.3) and is stored in the *fail_arr* variable.

The WEC requires retrieval if there are any failures which need that action (identified with the Boolean function *any_fails_need_retrieval*, section 7.14.6) or if there is any WEC-based maintenance due (using the Boolean function *any_maint_ready*, section 7.14.34). The task is only considered further is there is enough space at the O&M base (i.e. *if num_wecs_offsite < max_wecs_offsite*). The ID of the vessel to be used for the marine operation (*vessel_id_to_use*) is identified using the function *ret_vessel_id_to_use* (section 7.14.24) with *install_vessel_name* as an argument. If no suitable vessel is available then the returned value is -5. The code in this section only continues if a suitable vessel is found (i.e. *if vessel_id_to_use > 0*). The actual length of the marine operation is calculated by obtaining the vessel travel times (i.e. *get_free_travel_time + get_tow_time*) from the relevant *vessel* object (i.e. *vessel(vessel_id_to_use)*). This is added to the time it takes to retrieve the WEC once the vessel is at site (via *longest_time_offshore*, section 7.14.7) to give the value of *fuel_hours*. This value is rounded up to the nearest *time_step* to give the length of required marine operation in hours as a multiple of the number of intervals (*wndo_length*). The type of operational limits required for the retrieval operation, *ops_lim_type*, is calculated by the function *calmest_lims_for_op* (see section 7.14.8). Using this information, the accessibility of the weather window is assessed using the Boolean function *full_wndo_open* (section 7.14.11). The retrieval section code only continues if this function returns True, indicating that the weather window is open. The Boolean variable *delayed_by_techs* is then initialised to False, saying that the work is not delayed by a lack of technicians, before an *if* condition identifies whether external contractors cannot be hired (i.e. *if short_term_contracts_enabled = False*). If this is the case, then *delayed_by_techs* is changed to True if the number of available technicians (i.e. *technicians.get_num_techs_avail*, section 7.15) is less than the number of technicians required for the retrieval operation (i.e. same as an installation, *install_num_techs*). The retrieval operation is only carried out if this final condition is met (i.e. *if delayed_by_techs = False*). As soon as a constraint is met which causes the work to be delayed, then the *delays* object is updated by calling its subroutine *add_this_delay* (section 7.10.2) and setting the *delay_status* of the WEC accordingly.

If the retrieval operation is set to go ahead, then the identified vessel (*vessel_id_to_use*) is sent (*vessel.mobilise_boat*, section 7.8.4) for the number of intervals in the weather window (*wndo_length / time_step*). The ID of the vessel is stored in the variable *install_vessel_id_in_use* for use by the *next_interval* subroutine (see section 7.14.14). The number of contractors required for the retrieval operation (*num_contractors_needed*) is then calculated using the *install_num_techs* and the number of available technicians at the O&M base. If *num_contractors_needed* is zero, then the identifier of the number of O&M base technicians to assign to the task, *perm_techs_to_assign*, is set to be the value of *install_num_techs*. If contractors are needed, then all the available O&M base technicians are first stored in *perm_techs_to_assign*, before the contractors' fees are added (*technicians.add_contractor_fees*, section 7.15.3) for each of the *num_contractors_needed*. An error message is displayed if this point in the code is reached and the user has specified not to hire contractors (i.e. *if short_term_contracts_enabled = False*, section 7.15.1). Each of the O&M base technicians identified by *perm_techs_to_assign* is then assigned to the task by calling the subroutine *technicians.add_tech_working* (section 7.15.2). After technicians have been assigned, the WEC *state* is set to *being_removed* and the number of intervals required offsite to complete the repairs and/or maintenance tasks, *intervals_off_site*, is calculated by the function *calc_intervals_offsite* (see section 7.14.9). The number of intervals required to complete the retrieval

operation (*retrieval_ints*) is calculated by dividing the *wndo_length* by the *time_step*, and the *wec_power* is set to zero. An *If-Else* condition is then used to determine how to assign the vessel costs using the subroutine *assign_vessel_costs_output* (see section 7.14.12). This can either be sent the argument "maint_only" if only maintenance is going to be undertaken, "offsite repairs only" if the WEC only has failures, or "fails and maint" if both repairs and maintenance are going to be carried out. Finally, the *delay_status* of the WEC is set to "none" and any sustained failures which could have been repaired whilst the WEC was on site (i.e. by having parts replaced) are accounted for by trying to assign the necessary spare parts to the task (*try_assign_replacement_parts*, section 7.14.15). This assumes that the replacements can be made rapidly once the WEC is offsite and will not incur any additional time at the O&M base.

If the WEC does not require retrieval but there are failures (i.e. *if fail_arr(1) > 0*) then replacement of certain parts whilst the device is onsite (i.e. offshore) can take place. Initially, the number of intervals required to complete the task, *offshore_repair_time*, is set to zero. A list of actions (*actions_array*) required to correct the failures in *fail_arr* is obtained using the function *ret_actions_reqd* (section 7.14.25). This assumes that multiple failures requiring the same action can all be corrected at once by replacing the affected part. It is also assumes that two parts (i.e. a PTO unit and an instrumentation box) cannot be replaced in the same operation due to logistics surrounding vessel capacity. As a result, the action that takes priority (*this_action*) is calculated by the function *ret_action_onsite_priority* (section 7.14.22). The list of failures is then modified to contain only those faults which require the prioritised action (*new_fail_arr*). The name of the vessel required for this action is obtained by the function *vessel_for_action* (section 7.14.23) and stored as *vessel_name_reqd*. The ID of an available vessel with the identified name (*vessel_id_to_use*) is then found using the function *ret_vessel_id_to_use* (section 7.14.24). The availability of spare parts at the O&M base is assessed by first identifying the required part (*ret_part_to_replace*, section 7.14.10) then calling the function *all_parts_available* in the *parts* object (section 7.9.2). The code then continues with the same assessments seen in the retrieval section (i.e. *full_wndo_open*, technicians etc.) if the required spare part is available. If the marine operation is to go ahead after all of these constraints have been assessed, then it is simulated in a similar way as described previously in this section. The key differences between onsite repairs and WEC retrieval is that the *vessel_id_to_use* is set to be *onsite_vessel_id_in_use*, the *state* becomes *under_repair*, the *offshore_repair_time* is set to be *wndo_length / time_step* and *assign_vessel_costs_output* (see section 7.14.12) is sent the argument "onsite fails only". Also, the subroutine *order_new_parts* is called in the *parts* object (see section 7.9.2), thereby completing the start of the simulated 'onsite repair' marine operation.

7.14.5 Number of onsite technicians required

The function *num_onsite_techs_reqd* is used by the *attempt_fix* subroutines in both the *array_object* (section 7.13.4) and *wec_object* (section 7.14.4) class modules in order to identify the number of technicians required to undertake an onsite (i.e. offshore) repair (i.e. replacement of key parts). It is sent the arguments *fail_arr* (a list of failures), *vessel* (the *vessel_object* to access its procedures), and *this_vessel_id* (the ID of the vessel being used for the marine operation).

Firstly, the number of technicians required to fix the failures listed in *fail_arr* is calculated. This is achieved by initialising *num_techs_for_fails* to zero before a *for* loop considers each failure in turn using the identifier *i*. If the vessel required for that failures (i.e. *get_vessel_reqd* from the relevant *fail_param_list* object, section 7.3) matches the name of the vessel (i.e. *get_name* from the

vessel(this_vessel_id) object, section 7.8.2) then the number of technicians that the failure requires (i.e. *get_techs_reqd* from the relevant *fail_param_list* object, section 7.3) is added to the value of *num_techs_for_fails*. Once each failure has been considered, the value of *num_techs_for_fails* is checked against the capacity of the required vessel (i.e. *get_personnel_capacity* from the *vessel(this_vessel_id)* object, section 7.8.2). If more technicians are required than can fit on board the vessel, then the return value is set to the vessel capacity, thereby assuming that a full crew can undertake the operation. If the value of *num_techs_for_fails* is 1, then health and safety considerations forces the return value to become 2, stating that one technician can never undertake a marine operation alone. Error handling is in place to highlight if a vessel is given the capacity of 1 on the 'Vessels' spreadsheet (section 4.2). Otherwise, the return value is set to be *num_techs_for_fails*.

7.14.6 Any failures need retrieval

The Boolean function *any_fails_need_retrieval* is called by the *wec* procedures *attempt_fix* (section 7.14.4) and *next_interval* (section 7.14.14), as well as the cost-benefit analysis part of the model (section 7.12). It is used to identify if any of the failures listed in a *fail_arr* require the action "Retrieve WEC", as listed in the 'Inputs' spreadsheet (section 4.1.2).

The return value, *temp_bool*, is first initialised to False, saying that *fail_arr* doesn't contain any failures which require retrieval. If there any failures within the list (i.e. *if fail_arr(1) > 0*) then each one is considered in a *for* loop with the identifier *i*, from 1 to the upper boundary (i.e. *UBound*) of the *fail_arr*. If the action required to repair that fault (i.e. *get_action_reqd* from the relevant *fail_param_list* object, section 7.3) is "Retrieve WEC" then *temp_bool* is changed to True and the *for* loop is ended, thus avoiding any unnecessary looping. The function name, *any_fails_need_retrieval*, is set to *temp_bool*, becoming True if any of the failures listed in *fail_arr* do require WEC retrieval.

7.14.7 Longest time offshore

The Double function *longest_time_offshore* is used by the *wec* procedures *attempt_fix* (section 7.14.4) and *get_time_until_repaired* (section 7.14.29, involved with *cost_benefit_analysis*) in order to calculate the number of hours required to remove a WEC from site, given its failures and/or scheduled maintenance events. The function is sent the list of failures (*fail_arr*) and the action to be undertaken (*this_action*). The String variable *this_action* is either sent as "retrieve" if the WEC requires retrieval or "replace" if onsite repairs are going to take place.

The return value, *temp_long_time*, is first initialised to zero. If *this_action* is "retrieve" then the longest time required for WEC retrieval is first assessed from all the failures and then from all the maintenance events due. To achieve this, each of the failures (if there are any) is considered in a *for* loop. If the action required for that fault is "Retrieve WEC" (obtained using *get_action_reqd* from the relevant *fail_param_list* object, section 7.3) then *temp_long_time* is updated by taking the maximum value of the existing *temp_long_time* and the number of hours offshore required for that failure (i.e. *get_hours_offshore*) using the custom *max* function (see section 7.1.7). Note that this ignores any failures that could otherwise be repaired onsite because the model assumes these part replacement tasks can be completed easily whilst the WEC is at the O&M base for other repairs and/or maintenance. A new *for* loop then considers each WEC-based maintenance event listed in the *wec_maint_cat* variable (see section 7.14.1). If the event is scheduled to be undertaken (i.e. *this_maint_ready*, section 7.14.34) then *temp_long_time* is again updated using

the *max* function and *get_hours_offshore* from the relevant *maint_param_list* object (section 7.4). If the String variable *this_action* is given as "replace" then only failures need to be considered (because the WEC would have been set for retrieval otherwise). Each fault is considered in a *for* loop with the return value being updated every time by adding the existing *temp_long_time* to the number of hours required offshore (i.e. *get_hours_offshore* from the relevant *fail_param_list* object, section 7.3). The user is prompted to exit the program (i.e. *terminate_program*, section 7.1.8) if an invalid entry of *this_action* has been sent to the function. Otherwise, the function name (*longest_time_offshore*) is set to be the value of *temp_long_time* for use by the calling procedure.

7.14.8 Calmest limits for operations

The function *calmest_lims_for_op* is used to identify the ID of the type of operational limits with the most restrictive weather conditions from a list of failure and/or the scheduled maintenance events to be undertaken on the WEC. It is called the *wec* procedures *attempt_fix* (section 7.14.4) and *next_interval* (section 7.14.14). The functionality of *calmest_lims_for_op* shows the importance of the user listing the operational limits types in the correct order in the 'Ops Limits' spreadsheet, as described in section 4.4. The function is sent the list of failures (*fail_arr*) and the action to be undertaken (*this_action*). The String variable *this_action* is either defined as "retrieve" if the WEC requires retrieval, "replace" if onsite repairs are going to take place or "subsea" if array-based failures are being assessed.

The return value, *temp_min_ops_type*, is first initialised to 10 (i.e. a nominally large number). If *this_action* is "retrieve" then the failures requiring retrieval are assessed first, ahead of each scheduled maintenance event due to be carried out on the WEC. This follows a similar structure to the function *longest_time_offshore*, described in section 7.14.7, where failures and maintenance events are considered in a *for* loop. However, the difference is that the return value is updated by using the custom *min* function (not *max*, section 7.1.7) to find the minimum value of the existing *temp_min_ops_type* and the operational limits type of the task under consideration (*get_ops_limits_type*). If *this_action* is either "replace" or "subsea" then only the failures listed in *fail_arr* need to be assessed. Again, the custom function *min* is used to update the return value. Error handling is in place to prompt the user to end the program if the value of *this_action* is invalid, or if *temp_min_ops_type* is still equal to 10 (the initialised value) at the end of the procedure. Otherwise, the function name (*calmest_lims_for_op*) is set to be the value of *temp_min_ops_type* for use by the calling procedure.

7.14.9 Calculate intervals offsite

The function *calc_intervals_offsite* is called by the *wec* procedures *attempt_fix* (section 7.14.4), *next_interval* (section 7.14.14) and *get_time_until_repaired* (section 7.14.29) in order to calculate the number of intervals a WEC needs to spend offsite at the O&M base to undergo repairs and/or maintenance tasks. It is sent a list of the failures sustained by the WEC in the variable *fail_arr*.

The return value, *temp_intervals*, is first initialised to zero. Each of the failures contained within the *fail_arr* (if there are any) are then considered in a *for* loop with the identifier *i*. If the failure requires the action "Retrieve WEC" (obtained using *get_action_reqd* from the relevant *fail_param_list* object, section 7.3) then the number of days it requires at the O&M base is converted into intervals (i.e. *get_days_onshore * (24 / time_step)*) and added to the return value. A new *for* loop then considers each of the WEC-based maintenance tasks (in *wec_maint_cat*, defined in section 7.14.1). If the function *this_maint_ready* (section 7.14.34) identified that the task is due to be

completed, then the *temp_intervals* value is again updated by adding the number of days it requires at the O&M base (converted into intervals). Error handling is in place if the return value is still zero at the end of the procedure. Otherwise, the function name (*calc_intervals_offsite*) is set to be the value of *temp_intervals* for use by the calling procedure.

7.14.10 Part to replace

The String function *ret_part_to_replace* is called by the *wec* functions *attempt_fix* (section 7.14.4) and *try_assign_replacement_parts* (section 7.14.15) in order to identify which replacement parts (if any) are required to correct the failures sustained by the WEC. The list of failures is sent as the variable *fail_arr* and the *parts* object is also given so its procedures can be used in the function. The two calling procedures use *ret_part_to_replace* in slightly different ways. As described in section 7.14.4, the onsite repairs part of *attempt_fix* sends *ret_part_to_replace* a list of the failures which require the prioritised action (*new_fail_arr*). Therefore, only one replacement part will be identified. However, the procedure *try_assign_replacement_parts* does not assess the actions in the same way ahead of calling *ret_part_to_replace*. This is accounted for in the functionality of the procedure.

To achieve its purpose, the function first assesses the failures contained in *fail_arr* (if there are any) by sending it to the *parts* function *multi_replacement_types_arr* (see section 7.9.3) and storing the returned value in the variable array *parts_type_arr*. If the returned array contains only one String value, then that entry (i.e. *parts_type_arr(1)*) is considered. If it is "none" then the return value, *temp_ret*, is also set to "none". Otherwise, *temp_ret* becomes that entry's value. If more than one entry is contained in *parts_type_arr* then *temp_ret* is set to "multiple". If no failures have been sustained by the WEC, then the return value is set to "none" if scheduled maintenance is the reason why this function has been called (i.e. *if any_maint_due = True*, section 7.14.34). If neither repairs nor maintenance is to be carried out then this function should not have been called and the user is prompted to end the program (i.e. *terminate_program*, section 7.1.8). Otherwise, the function name (*ret_part_to_replace*) is set to be the value of *temp_ret* for use by the calling procedure.

7.14.11 Full window open

The Boolean function *full_wndo_open* is used to assess the accessibility of a weather window of a given length and severity. It is called whenever a marine operation is due to be undertaken in the model simulations; in the *wec* functions *attempt_fix* (section 7.14.4) and *next_interval* (section 7.14.14), as well as *attempt_fix* in the *array_object* class module (section 7.13.4). The function is sent the *weather* object to access its information and procedures, as well as the date information *irun* (current year) and *start_interval* (current interval in *irun*). The length of the required weather window is given in hours as *wndo_length* and the severity is identified by the type of operational limits in *ops_lims_type*.

The function name (*full_wndo_open*) is first initialised to False, saying that the required weather window is inaccessible (i.e. closed). The number of accessible intervals in the window, *count_open*, is initialised to zero. Each interval in the weather window is assessed in turn by using a *for* loop from 1 to the number of intervals represented by *wndo_length* (i.e. *wndo_length / time_step*) with the identifier *i*. If the interval being assessed occurs beyond the end of the array lifetime, then the window is deemed to be inaccessible. This is identified if *irun* is equal to the project lifetime (*no_run*) and if the interval being assessed (i.e. *start_interval - 1 + i*) goes beyond the number of

intervals in a year (*no_intervals*). In this case, the value of *count_open* is set to zero, effectively closing the weather window. If the interval under consideration is within the project lifetime, then the *weather* function *get_this_wndo* (see section 7.6.2) is used to determine if *this_wndo* is "OPEN" or "CLOSED". If *this_wndo* is "OPEN" (i.e. accessible), then 1 is added to the value of *count_open*. Once all intervals in the weather window have been considered, the Boolean value of *full_wndo_open* is only set to True if the value of *count_open* is equal to the number of intervals in the window (i.e. *wndo_length / time_step*), thereby telling the calling procedure that the full requested weather window is accessible.

7.14.12 Assign vessel costs output

The subroutine *assign_vessel_costs_output* is used throughout the model whenever a marine operation is started in order to assign the vessel costs (i.e. fuel and hire fees) to the appropriate failure categories and maintenance events. This occurs in the *wec* functions *attempt_fix* (section 7.14.4) and *next_interval* (section 7.14.14), as well as *attempt_fix* in the *array_object* class module (section 7.13.4). The information is used in producing the failures and maintenance output tables presented in the 'Results' spreadsheet (section 6.1). To achieve this, the subroutine is sent the incurred vessel costs (*total_hire_fees* and *total_fuel_cost*), a String identifier of how to assign the costs (*my_type*), a list of sustained failures (*fail_arr*), and the total number of hours required to complete the marine operation or offsite task (rounded up the nearest whole interval, *total_hours_loc*).

The String argument *my_type* is sent to the subroutine as one of the following options:

- "onsite fails only" - only on-site parts replacement tasks are taking place
- "offsite repairs only" - only off-site repairs are being undertaken
- "maint only" - only off-site scheduled maintenance is being undertaken
- "fails and maint" - scheduled maintenance and fault repairs are taking place

In the cases where *my_type* is given as either "onsite fails only" or "offsite repairs only", the function *fails_time_share_arr* is first used to create a two-dimensional array consisting of the ID of each failure (in the first dimension) and the corresponding portion of the costs to be assigned (in the second dimension, from 0 to 1), as described in section 7.14.13. This 2D array is stored using the variable *fails_share_arr*. Each entry of the array is considered in a *for* loop from 1 to *UBound(fails_share_ arr, 1)* (i.e. the number of failures) using the identifier *i*. The portion to be assigned to that failure (i.e. *fails_share arr(i, 2)*) is multiplied by the total costs incurred (*total_hire_fees* and *total_fuel_cost*) to calculate the share of the costs to be attributed (*hire_fees_attibuted* and *fuel_cost_attibuted* respectively). This information is then sent to the relevant function (either *set_vessel_hire_fees* or *set_vessel_fuel_cost*) in the failure output object (*fail_output_list*, see section 7.25.2) along with the ID of the failure (obtained from *fails_share_arr(i, 1)*).

If *my_type* is "maint only" then only the maintenance output object (*maint_output_list*) needs to be used. The vessel costs do not need to be attributed in as complex a way as the failure categories. Instead, the function *get_num_wec_maints_ready* (see section 7.14.34) is used to split the vessel costs evenly between the maintenance events being undertaken. This information is sent to the relevant procedure (*set_vessel_hire_fees* or *set_vessel_fuel_cost*) in *maint_output_list* (see section 7.26.2) along with the ID of the maintenance category (obtained from the *wec_maint_cat list*, see section 7.14.1).

Finally, if *my_type* is given as "fails and maint" then the portions of the costs to be assigned to the failure categories (*fail_portion*) and the maintenance events (*maint_portion*) must be calculated. Initially, the 2D array containing the shares attributed to each failure (*fails_share_arr*) is obtained using the function *fails_time_share_arr* (section 7.14.13). The total number of hours that the WEC needs to spend offsite (*total_hours_loc*) is separated into the time due to the maintenance events (*total_hours_maint*) and the time due to the failures (*total_hours_fails*). This is achieved by considering each of the WEC-based maintenance categories in a for loop and using the function *this_maint_ready* (see section 7.14.34) to identify whether that event is to be undertaken. If it is then the value of *total_hours_maint* is updated to include the number of hours that the event requires offsite (i.e. 24 multiplied by *get_days_onshore* from the relevant *maint_param_list* object, section 7.4). The final value of *total_hours_maint* is subtracted from *total_hours_loc* to identify the value of *total_hours_fails*. This method of calculating the hours a WEC requires offsite must match the functionality of the procedure *calc_intervals_offsite* (section 7.14.9). Next, the variables *fail_portion* and *maint_portion* are initialised to 0. If the corresponding number of hours (i.e. *total_hours_fails* and *total_hours_maint* respectively) is greater than zero then portion is calculated by dividing that value by *total_hours_loc*. The costs are then assigned by following the steps described in the previous paragraphs, with the only difference being that the attributed costs (i.e. *hire_fees_attributed* and *fuel_cost_attibuted*) are calculated by multiplying the share of the costs (e.g. *fails_share_arr(i, 2) * total_hire_fees* for failure category *i* or *total_hire_fees / get_num_wec_maints_ready* for a maintenance event, similar for fuel costs) by the corresponding portion (i.e. *fail_portion* or *maint_portion*).

7.14.13 Failures time share array

The function *fails_time_share_arr* is used to create a two-dimensional array containing the IDs of failures and their respective shares of the total number of hours required to complete the repairs. The function is called by the subroutine *assign_vessel_costs_output*, as described in section 7.14.13. It is sent a list of the sustained failures (*fail_arr*) as well as the String identifier *my_type*, which can be one of three possible options:

- "onsite fails only" - only on-site parts replacement tasks are taking place
- "offsite repairs only" - only off-site repairs are being undertaken
- "fails and maint" - scheduled maintenance and fault repairs are taking place

Firstly, the return value (*share_arr*) is first set up to be a two-dimensional VBA array where each entry in *fail_arr* is allocated two spaces (i.e. *ReDim share_arr(1 To UBound(fail_arr), 1 To 2)*). The total number of hours (*total_hours*) is also initialised to zero.

If *my_type* is given as "onsite fails only" then the shares are assigned based on the number of hours that each failure requires offshore (i.e. onsite) in order to replace the associated part. The total number of hours required offshore to repair these failures is calculated by adding each fault's *get_hours_offshore* value (obtained from the relevant *fail_param_list* object, section 7.3) to the *total_hours* in the same way as seen in the *wec* function *longest_time_offshore* (see section 7.14.7). The *share_arr* entries are then filled by looping through each failure and calculating the share of the total hours by dividing the *get_hours_offshore* by the *total_hours*. The calculated value *this_share* is placed in the second dimension of the *share_arr* whilst the ID of the failure (i.e. *fail_arr(i)*) is put in the first dimension.

If the value of *my_type* is either "offsite repairs only" or "fails and maint" then the share of the time is calculated based on the number of days required offsite at the O&M base to repair the failures. This follows the same process as the "onsite fails only" section with the primary difference being that *get_hours_offshore* (from the relevant *fail_param_list* object, section 7.3) is replaced by *get_days_onshore* multiplied by 24 (to convert days into hours). The value of *total_hours* is only updated if the failure under consideration does not have the *get_days_onshore* value of "N/A". In other words, when a WEC is being taken to the O&M base to undergo retrieval-based WEC failures or scheduled maintenance, then failures which could have been repaired onsite are not assigned any of the vessel costs. This assumes that such failures can be corrected easily and rapidly once the WEC is as the O&M base, as discussed previously (e.g. in section 7.14.7).

The function name (*fails_time_share_arr*) is set to be the value of *share_arr* for use by the calling procedure (*assign_vessel_costs_output*).

7.14.14 Next interval

The subroutine *next_interval* in each *wec* object is called by the procedure of the same name in the *array_object* class module, as described in section 7.13.7. It is used to set to *wec* up for the next interval and keep track of progress in repairs, maintenance and marine operations. If an offsite repair has been completed then *next_interval* assesses the constraints surrounding installation of the WEC and starts the marine operation as soon as possible. In order carry out its purpose, *next_interval* is sent the arguments *irun* (current year), *this_interval* (current interval in *irun*) and the number of WECs in the array (*num_wecs*), as well as the relevant objects *weather, vessel, technicians, parts* and *delays*. Several variables are used throughout the subroutine:

- *fail_arr()* - stores a list of WEC-based failures
- *i, j* - identifiers
- *fuel_hours* - actual number of hours to undertake a marine operation
- *wndo_length* - number of hours to undertake marine operation, rounded up to the nearest full *time_step*
- *offsite_hours* - the number of hours this WEC spent offsite for repairs / maintenance
- *this_tech* - identifier of technicians
- *current_fail_being_done* - current failure being repaired offsite
- *fails_array_position* - identifier of position in an array
- *current_maint_being_done* - current maintenance event being undertaken offsite
- *maint_array_position* - identifier of position in an array
- *actions_array()* - list of actions required to correct faults
- *this_action* - name of an action required to correct faults
- *new_fail_arr* - stores a modified list of WEC-based failures
- *vessel_name_reqd* - name of vessel required for marine operation
- *vessel_id_to_use* - ID of vessel to use for marine operation

The structure and functionality of the subroutine *next_interval* is shown in Figure 7.9. Limited procedures are identified in the flowchart, however, more detail is provided in the subsequent text.

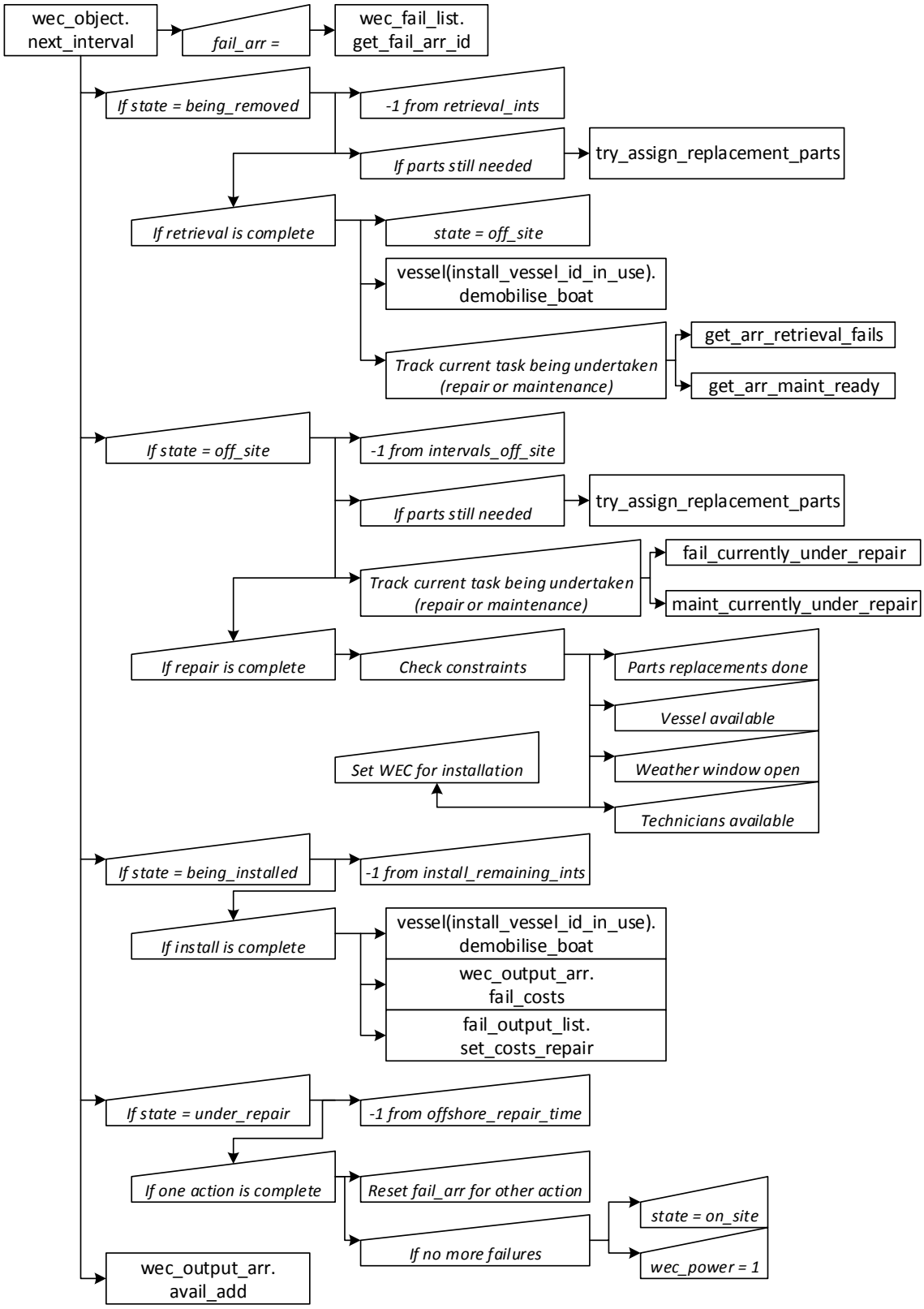

Figure 7.9. Structure of *next_interval* in the *wec_object* class module

As highlighted in Figure 7.9, the *next_interval* subroutine consists of four primary sections corresponding to the *state* of the WEC at the current interval; *being_removed, off_site, being_installed* and *under_repair*. The final lines of code in the procedure are used to check that the *wec_power* has not fallen below zero and update the 'availability' value for the 'Results' output spreadsheet (see section 6.1).

If the WEC *state* is *being_removed* then the *attempt_fix* procedure (section 7.14.4) has started to retrieval operation after assessing each of the constraints (such as weather conditions etc.). The *attempt_fix* subroutine set a number of variables that are used in *next_interval*. This includes the length of the marine operation in terms of the number of intervals, *retrieval_ints*. This is updated at each interval here by subtracting 1. The scenario where the WEC has been taken to the O&M base to undergo certain tasks whilst it has also sustained failures which could have been repaired offsite is accounted for by calling the *try_assign_replacement_parts* subroutine (see section 7.14.15). This is only called if the required parts have not already been assigned (identified with the Boolean variable *replacement_parts_delayed*). Once the retrieval operation has been completed (i.e. *if retrieval_ints <= 0*) then the WEC *state* is set to *off_site* and the vessel used can be demobilised (i.e. *vessel(install_vessel_id_in_use).demobilise_boat*, section 7.8.5). The value of *install_vessel_id_in_use* is then reset to zero, indicating that no vessel is currently assigned to this WEC. The tasks for which the WEC has been retrieved are undertaken immediately. This involves finding the list of failures which forced the retrieval (*offsite_failures_array*) using the function *get_arr_retrieval_fails* (section 7.14.16) and the list of due maintenance events (*offsite_maints_array*) using *get_arr_maint_ready* (section 7.14.17). For each of these lists, an array is created to store the number of intervals that has been spent working on each task so far (*offsite_fails_ints_worked* and *offsite_maint_ints_worked* respectively), with each entry being initialised to zero. The hierarchy of the code here means that failures are repaired ahead of maintenance being carried out, which is a realistic representation of the operations. Firstly, if there are failures, then the function *assign_offsite_fail_techs* (section 7.14.18) is called in order to assign technicians to the first repair task. If this function is successful (i.e. returns True) then the first entry in the offsite_*fails_ints_worked* array is changed to 1. Otherwise, the number of intervals required offsite to complete the WEC repairs and maintenance (*intervals_off_site*) is increased by 1. If there are no failures then the first maintenance event due is assigned the technicians it requires. This follows the same process as the repairs section, except that the function *assign_offsite_maint_techs* (section 7.14.18) is used, and the *offsite_maint_ints_worked* array is updated if successful. Error handling is in place if this section is utilised without either repairs or maintenance being required, or if the value of *install_vessel_id_in_use* is still zero.

If the WEC *state* is *off_site* then the previously discussed part of *next_interval* has identified that the retrieval operation has been completed and the arrays controlling the amount of worked undertaken so far (*offsite_fails_ints_worked* and *offsite_maint_ints_worked*) have been created. For each interval the *state* is *off_site*, the counter *intervals_off_site* is decreased by 1. As before, the subroutine *try_assign_replacement_parts* (see section 7.14.15) is called if some failures are still awaiting spare parts to be assigned (i.e. *if replacement_parts_delayed = True*). The ID of the failure currently being repaired (*current_fail_being_done*) is identified using the function *fail_currently_under_repair* (section 7.14.19). If this returns a value greater than zero then it means that a repair is still being carried out. The position of this failure in the *offsite_fails_ints_worked* array (*fails_array_position*) is obtained using the function *find_fails_array_position* (section 7.14.19). If the current interval is the first time this failure has been considered for this WEC's particular series of tasks, then the function *assign_offsite_fail_techs* (section 7.14.18) is again used to assign

technicians to that repair task. As before, the value of *intervals_off_site* is increased by 1 if there are not enough technicians available to start the task. If technicians have already been assigned to this failure, then the relevant position in the *offsite_fails_ints_worked* array is increased by 1. Once all failures have been repaired (i.e. the value of *current_fail_being_done* not greater than zero) then this same process is carried out for the maintenance tasks due on the WEC. The only difference is that the functions and variables names are related to maintenance, not the failures (i.e. *current_maint_being_done, maint_currently_being_done, maint_array_position, find_maint_array_position, offsite_maint_ints_worked, assign_offsite_maint_techs*, see section 7.14.19).

If the WEC *state* is *off_site* and the value of *intervals_off_site* is less than or equal to zero, then the WEC has been fully repaired and inspected and is ready for installation. This process follows the exact same functionality as the relevant part of the *attempt_fix* subroutine, where the retrieval marine operation is commenced, as described in section 7.14.4. The key difference is that the *delay_status* is changed to "parts" (and *delays.add_this_delay* called accordingly) if the WEC is still waiting for a new PTO unit or instrumentation box to be assigned (i.e. *if replacement_parts_delayed = False*). The only other differences from *attempt_fix* here are that the number of intervals required to complete the operation (*wndo_length / time_step*) is assigned to the variable *install_remaining_ints* (rather than *retrieval_ints*) and the *state* of the WEC is set to *being_installed*.

If the WEC *state* is *being_installed* then 1 is subtracted from the existing value of install_remaining_ints. Once this value is less than or equal to zero, then the installation operation has been completed. At this point, the vessel used for installation (*install_vessel_id_in_use*) is demobilised (i.e. *vessel(install_vessel_id_in_use).demobilise_boat*, section 7.8.5) and its value is reset to zero. The outputs costs are updated by calling the procedures *fail_costs* and *set_costs_repair* in the objects *wec_output_arr* (section 7.18.2) and *fail_output_list* (section 7.25.2) respectively. The WEC is reset to its original position with *state* set to *on_site*, the *start* function in *wec_fail_arr* being called (clearing the list of failures on the WEC, section 7.16.3), and the *wec_power* changed to 1 (i.e. full power). Each maintenance category that was completed whilst the WEC was offsite (i.e. *if this_maint_ready = True* for each WEC-based maintenance event, section 7.14.34) is also reset by changing the relevant entry of *maint_checker* (i.e. *maint_checker(irun, i)*) to 1 and *maint_due* (i.e. *maint_due(i)*) to 0.

If the WEC *state* is *under_repair* then a part is being replaced whilst the device is still on site, as defined in the subroutine *attempt_fix* (section 7.14.4). The value of *offshore_repair_time* is decreased by 1 at each interval. Once this value becomes less than or equal to zero, then the marine operation to replace the parts has been completed and the vessel is back at the O&M base. The vessel used for the operation (*onsite_vessel_id_in_use*) is demobilised (i.e. *vessel(onsite_vessel_id_in_use).demobilise_boat*, section 7.8.5) and its value is reset to zero. As discussed in section 7.14.4, the onsite repairs part of *attempt_fix* undertakes the tasks by prioritising one at a time using the function *ret_action_onsite_priority* (section 7.14.22). The selected action (*this_action*) is again identified from the *actions_array* (obtained using the function *ret_actions_reqd*, section 7.14.21). The list of failures is modified so that *new_fail_arr* contains only those failures which require *this_action*. This new list is then sent to the procedures *fail_costs* and *set_costs_repair* in the objects *wec_output_arr* (section 7.18.2) and *fail_output_list* (section 7.25.2) respectively in order to update the outputs only for the corrected failures. Then, rather than resetting the WEC-based failures object (*wec_fail_arr*) completely, its subroutine *update_fail_arr* (section 7.16.3) is called with *new_fail_arr* as the argument in order to clear only the corrected

failures. The situation where all the onsite repairs have been carried out is identified if the function *get_total_fails* from the *wec_fail_arr* object (section 7.16.3) returns the value of zero. In this case, the WEC *state* is set to *on_site* and the *wec_power* is reset to full (i.e. 1). However, if there are still WEC-based failures listed in the *wec_fail_arr* object, then *fail_arr* is used to store the list. Each of the failures is then considered in a *for* loop using the identifier *i*. The *wec_power* is updated throughout the loop by subtracting the power loss incurred, remembering to convert the array-based *get_power* from the relevant *fail_param_list* object (section 7.3) into *wec_power* format by multiplying it by *num_wecs*. The WEC *state* is reset to *on_site*, thereby leaving the WEC at the array site, but having corrected some of its sustained failures.

7.14.15 Try assigning replacement parts

The subroutine *try_assign_replacement_parts* is called by the *wec* procedures *attempt_fix* (section 7.14.4) and *next_interval* (section 7.14.14). As described in these sections, *try_assign_replacement_parts* is called when the WEC is being (and has been) retrieved and taken to the O&M base for other repairs or maintenance. It is used to identify whether any replacement parts are needed to correct failures which might have otherwise been repaired offshore (i.e. on site) and tries to assign them accordingly. It is sent the list of failures on the WEC (*fail_arr*) and the object *parts*. The Boolean identifier *replacement_parts_delayed*, defined for use throughout the *wec* class module (section 7.14), returns False if the parts are assigned successfully (or True otherwise).

Firstly, the function *ret_part_to_replace* (section 7.14.10) is called with the arguments *fail_arr* and *parts* in order to identify which, if any, parts need to be assigned. The String returned value is stored in the variable name *this_part_type* and, as described in section 7.14.10, can either return "none", "multiple" or the name of a part.

If *this_part_type* is found to be "none" then the identifier *replacement_parts_delayed* is set to False, telling the calling procedure that replacements are not delayed (because there are not any).

If *this_part_type* is found to be "multiple" then both part types are to be replaced. In this case, the function *multi_parts_types_available* (in the *parts* object, section 7.9.4) is used to identify whether all the required parts are available. If they are (i.e. *if multi_parts_types_available = True*) then the variable *parts_type_arr* stores a list of the replacement parts using the *parts* function *multi_replacement_types_arr* (section 7.9.3). Each of the entries in this list is then considered in a *for* loop with the identifier *i*. The name of the part (i.e. *parts_type_arr(i)*) is sent to the *part* subroutine *order_new_parts* (section 7.9.2) in order to assign it to the task and re-order a new one for delivery. The identifier *replacement_parts_delayed* is then set to False to tell other procedures in the *wec* object that there is not delay due to a lack of spare parts. On the other hand, if not all the parts are available (i.e. *if multi_parts_types_available = False*) then *replacement_parts_delayed* is set to True.

If *this_part_type* contains the name of a single part then the function *all_parts_available* (in the *parts* object, section 7.9.2) is used to identify whether that part is available. If the function returns True then the *part* subroutine *order_new_parts* (section 7.9.2) is called to assign the part to the WEC and re-order a new one for delivery. The identifier *replacement_parts_delayed* is then set to False. Otherwise, *replacement_parts_delayed* is set to True, thereby causing potential delays to installation of the WEC in *next_interval* (section 7.14.14).

7.14.16 Array of retrieval failures

The function *get_arr_retrieval_fails* is called during the *wec* subroutine *next_interval* (section 7.14.14) in order to identify which failures sustained by the WEC required retrieval. To achieve this, it uses the variable *fail_arr* to store a list of the sustained failures (from *wec_fail_arr.get_fail_arr_id*, section 7.16.3) and the identifier *retrieval_fails* (which is initialised to zero). If the WEC has sustained any failures then each entry in *fail_arr* is considered in a *for* loop using the identifier *i*. If the *action_reqd* for that failure (obtained from the relevant *fail_param_list*, section 7.3) is "Retrieve WEC" then the value of *retrieval_fails* is updated and the ID of the failure (i.e. *fail_arr(i)*) is added to the return value *temp_array*. If *retrieval_fails* is still zero at the end of the loop (i.e. no failures at all or no failures requiring retrieval) then the first and only entry of *temp_array* is set to be -5 (a nominal negative number). The function name *get_arr_retrieval_fails* is set to be equal to *temp_array* for use by the calling procedure (*next_interval*).

7.14.17 Array of due maintenance categories

The function *get_arr_maint_ready* is called during the *wec* subroutine *next_interval* (section 7.14.14) in order to identify which maintenance events are currently due to be carried out on the WEC. The identifier *count_maint* is first initialised to zero. Each of the WEC-based maintenance categories (in *wec_maint_cat*, see section 7.14.1) is assessed in a *for* loop. If the function *this_maint_ready* returns True for that event then *count_maint* is updated and the ID of the maintenance category (i.e. *wec_maint_cat (i)*) is added to the return value *temp_array*. If *count_maint* is still zero at the end of the loop (i.e. no maintenance is due) then the first and only entry of *temp_array* is set to be -5 (a nominal negative number). The function name *get_arr_maint_ready* is set to be equal to *temp_array* for use by the calling procedure (*next_interval*).

7.14.18 Assign offsite technicians

The two Boolean functions *assign_offsite_fail_techs* and *assign_offsite_maint_techs* are both called during the *wec* subroutine *next_interval* (section 7.14.14) to identify if there are enough available technicians to undertake the current repair or maintenance task (respectively). If the required technicians are available or if external contractors can be hired, then the functions assign the relevant O&M base technicians to the task and update the contractor fees incurred if necessary. The functions are sent the arguments of *irun* (current year), *this_interval* (current interval in *irun*), the *technicians* object and *this_cat* (the ID of the failure category or maintenance event being undertaken). The two functions have exactly the same functionality and utilise the same procedures. The only difference is that *assign_offsite_fail_techs* accesses information for the failure using the *fail_param_list* class module (i.e. *With fail_param_list.get_fail_param(this_cat)*, section 7.3), whilst *assign_offsite_maint_techs* utilises the *maint_param_list* object (i.e. *With maint_param_list.get_maint_param(this_cat)*, section 7.4).

Firstly, the Boolean return value (*temp_bool*) is initialised to False, meaning that the work cannot go ahead due to a lack of available technicians. The next section of the functions then assess whether the work is going to be delayed. Another Boolean variable, *delayed_by_techs*, is initialised to False, saying that the work can go ahead. However, if the user has specified in the 'Labour' spreadsheet (see section 4.3) that external contractors cannot be hired (i.e. *if short_term_contracts_enabled = False*) the only the O&M base technicians can be used. If the number of available technicians (identified using the function *get_num_techs_avail* in the *technicians* object, see section 7.15.6) is less than the required number of technicians for that task (i.e. *get_techs_*

reqd) then the Boolean value of *delayed_by_techs* is changed to True. If contractors can be hired then *delayed_by_techs* stays as False.

The second part of the functions then assigns technicians to the task and updates contractor fees, only if *delayed_by_techs* is False. The return value *temp_bool* is first changed to True so that the calling procedure knows that the work can go ahead. The number of contractors required (*num_ contractors_needed*) is then identified by subtracting the number of available technicians (i.e. *technicians.get_num_techs_avail*, section 7.15.6) from the number of technicians required for the task under consideration (i.e. *get_techs_reqd*). If there is at least 1 contractor required then the number of O&M base technicians to be assigned (i.e. *perm_techs_to_assign*) is set to be the number of available technicians. Each of the *num_contractors_needed* is then considered in a *for* loop and the technicians output is updated by calling *add_contractor_fees* in the *technicians* object (see section 7.15.3), with the costs to be assigned at the next interval (i.e. *this_interval + 1*) for the duration of the task (i.e. *get_days_onshore* converted into intervals by multiplying by (24 / *time_ step*)). If *num_contractors_needed* is greater than zero but the value of *short_term_contracts_ enabled* is False then the user is informed of the error. If no contractors are needed then the variable *perm_techs_to_assign* is simply assigned to be the value of *get_techs_reqd*.

Each of the specified O&M base technicians to assign then considered in a *for* loop, from 1 to *perm_techs_to_assign*, using the identifier *j*. For each value of *j*, a nested *for* loop looks at every O&M base technician (from 1 to *num_technicians*) using the identifier *this_tech*. If *this_tech* is available (identified using the *technicians* function *get_tech_availability*, see section 7.15.6) then the person is assigned to the task by calling the *technicians* subroutine *add_tech_working* (section 7.15.2) with the arguments of *this_tech* and the *get_days_onshore* (converted into intervals by multiplying by (24 / *time_ step*)). The nested *for* loop is exited once an available technician has been founded for that entry of *perm_techs_to_assign*.

The function name (*assign_offsite_fail_techs* or *assign_offsite_maint_techs*) is set to be equal to Boolean return value *temp_bool* for use by the calling procedure (*next_interval*).

7.14.19 Offsite tasks array

When a WEC has been retrieved from site it is taken to an onshore or quayside O&M base to be repaired or inspected. The VBA code must keep track of which repairs and maintenance tasks have been completed so that technicians can be assigned to the work as appropriate for the required length of time. This aspect is simulated by the *wec* procedure *next_interval*, as described in section 7.14.14), by utilising the following four functions:

- fail_currently_under_repair - ID of failure being repaired
- find_fails_array_position - position of a certain failure in *offsite_failures_array*
- maint_currently_being_done - ID of maintenance task being undertaken
- find_maint_array_position - position of a certain maintenance event in *offsite_ maints_array*

The function *fail_currently_under_repair* utilises the two offsite-failures variable arrays defined for the whole *wec* class module, *offsite_failures_array* and *offsite_fails_ints_worked*, in order to find which failure is currently being repaired. The return value (*this_fail*) is first initialised to -5, saying that no failures are currently being repaired. If the WEC has had failures to repair (i.e. *if offsite_ failures_array(1) > 0*) then each entry of *offsite_failures_array* is considered in a *for* loop with the identifier *i*. If the number of intervals currently worked on that failure (i.e. *offsite_fails_ints_*

worked(i)) is less than the number required to finish the repair (i.e. *get_days_onshore* from the relevant *fail_param_list*, section 7.3, converted into intervals) then the return value *this_fail* is set to be the ID of that failure (i.e. *offsite_failures_array(i)*) and the loop is ended. The function name *fail_currently_under_repair* is set to be equal to *this_fail* for use by the calling procedure (*next_ interval*).

The function *find_fails_array_position* is used to find the position of a failure ID (*this_fail*) in the variable *offsite_failures_array*. To achieve this, the return value *temp_val* is initialised to zero before a *for* loop considers each entry in *offsite_failures_array* using the identifier *i*. If the entry under consideration matches the value of *this_fail* then *temp_val* is set to *i* and the *for* loop is ended. The function name *find_fails_array_position* is set to be equal to *temp_val* for use by the calling procedure (*next_ interval*).

The function *maint_currently_being_done* utilises the two maintenance-based variable arrays defined for the whole *wec* class module, *offsite_maints_array* and *offsite_maint_ints_worked*, in order to find which maintenance event is currently being undertaken. The return value (*this_maint*) is first initialised to -5, saying that no maintenance is currently being done. If the WEC has had maintenance to do (i.e. *if offsite_maints_array (1) > 0*) then each entry of *offsite_maints_array* is considered in a *for* loop with the identifier *i*. If the number of intervals currently worked on that maintenance event (i.e. *offsite_maint_ints_worked (i)*) is less than the number required to finish the task (i.e. *get_days_onshore* from the relevant *maint_param_list*, section 7.4, converted into intervals) then the return value *this_maint* is set to be the ID of that maintenance event (i.e. *offsite_maints_array (i)*) and the loop is ended. The function name *maint_currently_being_done* is set to be equal to *this_maint* for use by the calling procedure (*next_ interval*).

The function *find_maint_array_position* is used to find the position of the ID of a maintenance event (*this_maint*) in the variable *offsite_maints_array*. To achieve this, the return value *temp_val* is initialised to zero before a *for* loop considers each entry in *offsite_maints_array* using the identifier *i*. If the entry under consideration matches the value of *this_maint* then *temp_val* is set to *i* and the *for* loop is ended. The function name *find_maint_array_position* is set to be equal to *temp_val* for use by the calling procedure (*next_ interval*).

7.14.20 Print interval

The subroutine *print_interval* prints information about the WEC at every interval of the model lifetime to the 'run sheets' spreadsheets (section 6.2) in a 'full run' process is taking place, as described in section 5.2. The subroutine is called for each *wec* by *print_interval* in the *array_object* class module (section 7.13.12). It is sent the name of the output sheet (*run_sh*), the current interval (*this_interval*) and the first column to start printing for this WEC (*first_print_col*).

The headers are printed if the current interval is the first of the year (i.e. *if this_interval = 1*). This includes the ID of the WEC (using "WEC " and *wec_id*), as well as "failures" for the first column and "maintenance" for the second column. In addition, the overall section header "WECs" is printed if the *wec_id* is 1.

The first column of the WEC's printed section covers the sustained failures and repairs. A String list of the sustained failures (*string_list*) is obtained using the *string_fail* function in the *wec_fail_arr* object (section 7.16.3). Each of the possible WEC states is then considered with *If-Else* conditions. If the *state* is *being_removed* then the cell in the failures columns will read "Being removed" and will

show the list of sustained failures (*string_list*). If the *state* is *being_installed* then the cell will read "Being installed" without the failures information (because they have all been repaired by that point). In both these transit cases, the cell is filled grey. If the *state* is *under_repair* then the cell reads “Under repair” and shows the *string_list*, as well as being filled dark red. If the *state* is *off_site* then the cell reads “Off site” and also shows the *string_list*, as well as being filled dark blue. If none of these conditions are met then the WEC *state* must be *on_site*. In this case, the failures cell just shows the list of failures (*string_list*) and is filled with the colour matching the most severe class of fault sustained. The severity information is obtained by setting the returned value of the function *get_fail_number* in the *wec_fail_arr* object (section 7.16.3) to the variable *fail_number*. This is defined as a *New failure_no_object* class module (section 7.16.5), containing information about the classification of faults. If there are failures on the *on_site* WEC (i.e. if *string_list* contains any failure IDs) then the cell is filled red if the most severe failure is ‘major’, amber if ‘intermediate’, and green if ‘minor’.

The second column of the WEC’s printed section contains information about the maintenance events scheduled on the device. If no maintenance is due (i.e. *if any_maint_due = False*, see section 7.14.34) then the cell will simple read “Not due” with no background fill. If one or more maintenance events is due on the WEC, however, then a *string_list* is created to contain their IDs. This is achieved by looping through each of the WEC-based maintenance categories with the identifier *i* and using the variable *maint_due* to find whether that event is due (i.e. *if maint_due(i) = 1*). If so, then a series of *If-Else* conditions control the formatting of the *string_list* to make it readable. After the loop, the user-defined input about the amount of space available at the O&M base for WECs solely undergoing maintenance is utilised by checking the function *any_maint_delay* (section 7.14.35). If the value is False then the cell prints “Due” and the *string_list*, as well as being coloured red, indicating that the listed maintenance events are to be carried out. If *any_maint_ delay* is True then the cell will read “Delay retrieval for maint” with the *string_list* and is coloured amber, indicating that the listed maintenance events are scheduled but are delayed due to a lack of space at the O&M base.

7.14.21 Return actions required

The function *ret_actions_reqd* is called on multiple occasions throughout the model in order to identify the actions required to correct any failures listed in *fail_arr* whilst the WEC is on site (i.e. offshore). It is called by the *array_object* procedures *attempt_fix* (section 7.13.4), *next_interval* (section 7.13.7) and *assign_lost_revenue_fails_maint* (section 7.13.8), the *wec_object* procedures *attempt_fix* (section 7.14.4) and *next_interval* (section 7.14.14), and the *cost_benefit_analysis* function *worth_repairing_WEC* (section 7.12.4).

Firstly, the return value *temp_array* is set up to be a one entry array containing the String “n/a”. A counter (*count*) is also initialised to 1. If failures have been sustained then each one is considered using a *for* loop with the identifier *i* (from 1 to *UBound(fail_arr)*). The action required for that failure is obtained from the *get_action_reqd* function from the relevant *fail_param_list* object (section 7.3) and stored in the variable *this_action*. If it is the first time in the loop that any action has been considered then the first entry of the *temp_array* is changed to equal *this_action*. Otherwise, the custom function *is_in_array* (section 7.1.9) is used to determine whether *this_action* already exists in *temp_array*. If it doesn’t (i.e. if *is_in_array* = False) then 1 is added to the value of *count* and that entry of *temp_array* is changed to be *this_action*. An error message is displayed if there are no failures (i.e. if *fail_arr(1)* is less than or equal to zero and the user is

prompted to exit the program (*terminate_program*, section 7.1.8). Another error message is displayed if the first entry of *temp_array* is still "n/a" at the end of the function. The function name, *ret_actions_reqd*, is set to be *temp_array* so it can be used by the calling procedures.

7.14.22 Return onsite action priority

The String function *ret_action_onsite_priority* is used to identify the onsite replacement action (from a list of possibilities, *actions_array*) which should take priority from the current list of sustained failures (*fail_arr*). It is called by the *array_object* procedure *assign_lost_revenue_fails_maint* (section 7.13.8), the *wec_object* procedures *attempt_fix* (section 7.14.4) and *next_interval* (section 7.14.14), and the *cost_benefit_analysis* function *worth_repairing_WEC* (section 7.12.4).

To achieve this, *ret_action_onsite_priority* first initialises the counter *max_count* to zero and sets the return value *temp_priority* to "n/a". Each action listed in the *actions_array* is then considered in a *for* loop with the identifier *i*. For each one, another counter (*count*) is initialised to zero. A nested *for* loop then considers every failure listed in *fail_arr* and identifies the action required to correct it using the *get_action_reqd* function from the relevant *fail_param_list* object (section 7.3). If this action matches the String value listed in the *actions_array* (i.e. *actions_array(i)*) then 1 is added to the value of *count*. After all the failures have been considered for that entry in *actions_array* then *temp_priority* becomes *actions_array(i)* if the value of *count* is greater than the existing value of *max_count*. An error message is displayed if the value of *temp_priority* is still "n/a" at the end of the function. The function name, *ret_action_onsite_priority*, is set to be *temp_priority* so it can be used by the calling procedures.

7.14.23 Vessel for action

The String function *vessel_for_action* is used to identify the name of a vessel suitable for undertaking a given onsite replacement action (*this_action*) based on a list of failures sustained by the WEC (fail_arr). It also undertakes error handling to ensure that the onsite placements aspect of the model is operating as planned. The function is called by the *array_object* procedure *attempt_fix* (section 7.13.4) and the *wec_object* procedures *attempt_fix* (section 7.14.4) and *get_time_until_repaired* (section 7.14.29).

The counter to track the number of failures requiring *this_action* (*count_fails*) is first initialised to zero. A *for* loop then considers each fault listed in the *fail_arr* with the identifier *i*. If the action required to correct that failure (i.e. *get_action_reqd* from the relevant *fail_param_list* object, section 7.3) matches *this_action* then 1 is added to the value of *count_fails*. For the first time *count_fails* is updated, the return String value *vessel_name* is set to be the name of the vessel required by that failure (i.e. *get_vessel_reqd* from the relevant *fail_param_list* object, section 7.3) and the variable *count_vessel_match* is initialised to 1. For every other value of *count_fails* identified throughout the loop, the variable *temp_name* is used to store the name of the vessel required by the failure under consideration. If *temp_name* is the same as the first vessel identified (vessel_name) then 1 is added to the value of *count_vessel_match*. This method ensures that failures requiring the same action also need the same vessel, as listed in the 'Inputs' spreadsheet (section 4.1.2). If *count_fails* is still equal to zero after the failures loop then an error message is displayed explaining that no failure requiring *this_action* has been found. Another error message is displayed if the two counters (*count_fails* and *count_vessel_match*) do not match explaining that two or more different vessels have been identified for the same action. In both cases, the user is

prompted to exit the program (*terminate_program*, section 7.1.8). If the function is successful then *vessel_for_action* is set to be *vessel_name* to be read by the calling procedure.

7.14.24 Return ID of vessel to use

The Integer function *ret_vessel_id_to_use* is used to identify the ID of an available vessel of a given name (*vessel_name*) and utilises the *vessel* class module. It is called by the *array_object* subroutine *attempt_ fix* (section 7.13.4) as well as the *wec_object* procedures *attempt_fix* (section 7.14.4) and *next_interval* (section 7.14.14).

The return value, *vessel_for_op*, is first initialised to a nominal negative number (-5) before a *for* loop considers each vessel (1 to *num_vessels*) listed in the 'Vessels' spreadsheet (section 4.2) with the identifier *i*. If the name of the vessel under consideration (i.e. *vessel(i).get_name*, section 7.8.2) matches the *vessel_name* and it is not already being used for a marine operation (i.e. *if vessel(i) .get_state = not_in_use*) and it passes the availability test (i.e. *if vessel(i).check_availability = True*, section 7.8.3) then the return value (*vessel_for_op*) is changed to be the vessel's ID (*i*) and the *for* loop is ended. The function name, *ret_vessel_id_to_use*, is set to be *vessel_for_op* so it can be used by the calling procedures.

7.14.25 Return action failures

The Integer array function *ret_action_fails* is used to create a list of failure IDs which require a particular repair action (*this_action*) from the options in an existing list (*fail_arr*). It is called by the *array_object* procedure *assign_lost_revenue_fails_ maint* (section 7.13.8), the *wec_object* procedures *attempt_fix* (section 7.14.4) and *next_interval* (section 7.14.14), and the *cost_benefit_ analysis* function *worth_repairing_WEC* (section 7.12.4).

The number of failures requiring *this_action* (*count*) is first initialised to zero before a *for* loop considers each failure in the *fail_arr* with the identifier *i*. If the action required to correct that failure (i.e. *get_action_reqd* from the relevant *fail_param_list* object, section 7.3) matches *this_action* then 1 is added to the value of *count*. The return array *temp_arr* is then re-sized to capture the new value of *count* and the entry is filled with the ID of the failure (i.e. *fail_arr(i)*). The function name, *ret_action_fails*, is set to be *temp_arr* so it can be used by the calling procedures.

7.14.26 Get total costs

The two Double functions *get_total_parts_costs* and *get_total_other_costs* are called by the *cost_ benefit_analysis* object (section 7.12) in order to obtain the total parts and other costs respectively that will be incurred when all the failures listed in *fail_arr* are repaired. The function achieve this by initialising the return value, *this_sum*, to zero before looping through all entries of the *fail_arr* and adding the relevant cost (obtained via the *fail_param_list* class module, section 7.3) to the value of *this_sum*. This value is then set to the function name for use by the calling procedures.

7.14.27 Maximum severity of failures

The function *get_max_severity* has the data type 'severity' (see section 7.1.1) and is called during the *cost_benefit_analysis* (section 7.12) in order to obtain the maximum severity of a list of sustained failures (*fail_arr*). It achieves this by initialising the return value (*temp_sev*) to *minor* before looping through all entries of the *fail_arr* with the identifier *i*. The *get_severity* function from the corresponding *fail_param_list* object (section 7.3) then identifies whether the failure is

classed as *intermediate*. If so, then the value of *temp_sev* is changed to *intermediate*. However, if the identified severity is *major* then *temp_sev* is changed accordingly but the loop is then exited, thereby making sure that the maximum severity is found. The value of *get_max_severity* is set to be *temp_sev* for use by the calling procedures.

7.14.28 Major and intermediate failures

The Boolean functions *major_failures* and *intermediate_failures* are utilised by the *order_this_list* procedure in the *cost_benefit_analysis* object (section 7.12.5) in order to identify if there are any failures in a list (*fail_list*) with the classification *major* and *intermediate* respectively. In both functions, the return value *temp_bool* is initialised to False, saying that no such failures have been found in the list. Each failure is then considered in a *for* loop. If the severity of the failure matches that indicated by the function name (i.e. *major* or *intermediate*) then *temp_bool* is changed to True and the loop is ended. The function name is then set to be *temp_bool* so it can be used by the calling procedure (*order_this_list*).

7.14.29 Time until repaired

The function *get_time_until_repaired* is used to calculate the number of intervals it takes to undertake a certain marine operation or offsite repair/maintenance task and also updates the total vessel cost incurred during the operation. It is called by the *cost_benefit_analysis* procedures *worth_retrieving_WEC* (section 7.12.3), *worth_repairing_WEC* (section 7.12.4) and *order_this_list* (section 7.12.5). The function is sent the argument *this_interval* (the current interval), *fail_arr* (a list of sustained failures), the *vessel* object, *condition* (a identifier of the type of operation being assessed) and *this_action* (a String variable indicating the action required, only applicable if onsite replacement of parts is being assessed). The *total_vessel_cost* is also sent to the function as a *ByRef* type, meaning that it can be modified by *get_time_until_repaired*. The *condition* variable is sent to the function as either "retrieval", indicating that the WEC under assessment would need to be retrieved for repairs and maintenance at the O&M base, or “onsite”, saying that the failures could be corrected by replacing the relevant parts whilst the WEC is onsite (i.e. offshore).

If the condition is “retrieval” then the function is used calculate the number of intervals from the time the vessel is set for the retrieval operation until the WEC is ready to be installed following offsite repair. The ID of the vessel to use for the operation (*vessel_id*) is identified by the *find_install_vessel_id* function (section 7.14.31). The length of the retrieval operation in hours (*fuel_hours*) is then calculated by summing the travel times of the vessel from the O&M base to site (without towing, *get_free_travel_time*) and back again (with towing, *get_tow_time*) using the relevant *vessel* object (i.e. *vessel(vessel_id)*, section 7.8.2), as well as the returned value of the function *longest_time_offshore* (section 7.14.7). The value of *fuel_hours* is then rounded up to the whole *time_step* and stored in the variable *wndo_length*. The vessel costs incurred during the retrieval operation (*temp_cost*) are calculated by utilising the functions *calc_hire_fees_for_op* (section 7.8.6) and *calc_fuel_for_op* (section 7.8.7) from the *vessel* object. The number of intervals the WEC will need to spend offsite for repairs and/or maintenance (*ints_offsite*) is calculated by the function *calc_intervals_offsite* (section 7.14.9), assuming no delays occur. The return value *temp_time* is set to be the *wndo_length* converted into intervals (i.e. by dividing by *time_step*) plus the value of *ints_offsite*.

If the condition is “onsite” then the function is used to calculate the number of intervals from the time the vessel sets off for site until the time it returns to the O&M base, having completed the

onsite replacement of parts task. The name of the vessel required for *this_action* (*vessel_name_reqd*) is identified by the function *vessel_for_action* (section 7.14.23). The ID of a vessel with this name (*vessel_id*) is the identified in a *for* loop from 1 to the number of vessels listed (*num_vessels*). Using the *vessel* object for the *vessel_id*, the length of the operation in hours (*fuel_hours*) is calculated by summing two vessel trips (without towing, *get_free_travel_time,* section 7.8.2) and the value obtained from the function *longest_time_offshore* (section 7.14.7). As before, this enables the vessel costs (*temp_cost*) to be identified by utilising the *vessel* functions *calc_hire_fees_for_op* (section 7.8.6) and *calc_fuel_for_op* (section 7.8.7). The value of *fuel_hours* is rounded up to the nearest whole interval (*wndo_length*) enabling the return value (*temp_time*) to be set to the number of intervals is takes to complete the marine operation (i.e. *wndo_length / time_step*)

Error handling is in place in the value of *condition* is invalid. If the function is a success then the *total_vessel_cost* is returned to the calling procedure as *temp_cost*, and *get_time_until_repaired* becomes *temp_time*.

7.14.30 Installation time

The function *get_install_time* is used to calculate the number of intervals it would take to complete an installation operation of a WEC and also updates the vessel cost incurred. It is called by the *cost_benefit_analysis* procedures *worth_retrieving_WEC* (section 7.12.3) and *order_this_list* (section 7.12.5). The function is sent the arguments *this_interval* (the current interval), the *vessel* object and *total_vessel_cost* (as a *ByRef* type so that it can be modified).

The return value of the vessel cost (*temp_cost*) is first initialised to be the value of *total_vessel_cost* sent by the calling procedure. The ID of the vessel to use for the operation (*vessel_id*) is then identified by the *find_install_vessel_id* function (section 7.14.31). The length of the installation operation in hours (*fuel_hours*) is calculated by summing the travel times of the vessel from the O&M base to site (with towing, *get_tow_time*) and back again (without towing, *get_free_travel_time*) using the relevant *vessel* object (i.e. *vessel(vessel_id)*, section 7.8.2), as well as the value of *install_hours* defined during the *start* subroutine (section 7.14.1). The value of *fuel_hours* is then rounded up to the whole *time_step* and stored in the variable *wndo_length*. The vessel costs incurred during the installation operation (*temp_cost*) are updated by utilising the functions *calc_hire_fees_for_op* (section 7.8.6) and *calc_fuel_for_op* (section 7.8.7) from the *vessel* object. The return value *temp_time* is then set to be the *wndo_length* converted into intervals (i.e. by dividing by *time_step*). The *total_vessel_cost* is returned to the calling procedure as *temp_cost*, and the function name (*get_install_time*) becomes *temp_time*.

7.14.31 Find installation vessel ID

The function *find_install_vessel_id* is used to obtain the ID of any vessel whose name matches the *install_vessel_name* defined during the *wec* subroutine *start* (section 7.14.1). It is utilised by the two procedures *get_time_until_repaired* and *get_install_time* described previously in sections 7.14.29 and 7.14.30 respectively. The function is sent the *vessel* object as an argument so that it can access the function *get_name* (section 7.8.2). A *for* loop considers each vessel listed in the 'Vessels' spreadsheet (i.e. from 1 to *num_vessels*) using the identifier *i*. If the name of the vessel under consideration matches the value of *install_vessel_name* then the return value *temp_id* is set to the ID of that vessel (i.e. *i*) and the *for* loop is ended. The function name *find_install_vessel_id* is set to be *temp_id* to be recognised by the calling procedures.

7.14.32 Intervals to next maintenance

The function *ints_to_next_maint* is used to calculate the number of intervals until the WEC will be due for its next scheduled maintenance event. It is called by the *cost_benefit_analysis* procedures *worth_retrieving_WEC* (section 7.12.3) and *worth_repairing_WEC* (section 7.12.4). It is sent the date information in the form *irun* (current year) and *current_int_long* (the current interval independent of the year. It is also sent the number of intervals the WEC will not be operating at site (*time_not_onsite*), which corresponds to the time it takes to carry out the marine operation (as well as the offsite work, if applicable).

The function is called for two different cases; one assessing the scenario where the WEC is repaired, and the other assessing the scenario where the WEC is left operating at site. These cases are identified by the *condition* being set as either "after repair" or "if left" respectively. If the *condition* is "after repair" then the interval to start the calculations from (*long_condition_interval*) is identified as the *current_int_long* plus the *time_not_onsite*. If *condition* is "if left" then the *time_not_onsite* is ignored so that *long_condition_interval* is simply the value of *current_int_long*.

The return value *temp_time* is first initialised to be the total number of intervals in the project lifetime (i.e. *no_intervals* multiplied by *no_run*). Each WEC-based maintenance event (listed in *wec_maint_cat*, see section 7.14.1) is then considered in a *for* loop using the identifier *imaint*. The time calculated by the previous entry (or the initialised value if this is the first entry in the loop) is stored in the variable *prev_time* by setting it to *temp_time*. The interval in the year where that maintenance event will be due, as defined by the season specified by the user on the 'Inputs' spreadsheet (section 4.1.3), is identified using the function *get_maint_interval_in_year*. This function uses the ID of the maintenance event (*this_cat*) to obtain the value of *get_time_of_year* from the relevant *maint_param_list* (section 7.4). It then converts this into interval format by identifying the first day of the first month of that season (based on the meteorological definition: winter is 1st December, spring is 1st March, summer is 1st June, and autumn 1st September). The value returned by the function *get_maint_interval_in_year* is stored in the variable *maint_due_ interval*. The function *get_maint_interval_in_year* is also used by the procedure *determine_fix* in the *array_object* class module, as described in section 7.13.3.

The variable *maint_checker* is then used to see if that maintenance category (*imaint*) has not already been completed in that year. This is indicated if the relevant entry in *maint_checker* (i.e. *maint_checker(irun, imaint)*) is zero. The current interval, when converted into a year-dependent interval (i.e. *current_int_long - ((irun - 1) * no_intervals)*), should be less than the value of *maint_ due_interval*. An error message is displayed if this is not the case. Otherwise, the variable *long_ condition_interval* is also checked to see if the maintenance category will become due during the time spent undertaking the marine operation and repair. If so, then the return value (*temp_time*) is set to -5, telling the calling procedures to delay the repair until maintenance is due. If not then the *temp_time* is simply set to be the number of intervals from the current interval (*long_condition_ interval*, converted into a year-dependent format) until the *maint_due_ interval*.

However, if the relevant entry of *maint_checker* is equal to 1 then either that maintenance event has already been completed in the year *irun*, or it was not scheduled to be undertaken in that year anyway. The year in which *long_condition_interval* occurs is calculated by dividing it by the number of intervals in a year (*no_intervals*) and rounding up to the nearest whole number, giving the value of *temp_year*. If the current year (*irun*) is the last year of the project lifetime (*no_run*), or if the

repair will go beyond the project lifetime (i.e. *if temp_year > no_run*), then the repair is delayed (*temp_time = -5*). If neither of these conditions is met then the next year when maintenance will be due (*maint_due_year*, initialised as zero) is identified using a *for* loop. The loop checks all remaining years of the project lifetime (from *irun+1* to *no_run*, with the identifier *i*) and sets *maint_due_year* to be that year if the relevant entry of *maint_checker* (i.e. *maint_checker(i, 1)*) is zero. If the value of *maint_due_year* is still returned as zero then that maintenance event will not be due again before the project lifetime ends. In this case, the interval when the WEC will be installed or finished onsite repairs (*install_int*, in year-dependent format) is used to calculate the number of interval left until the project ends (i.e. *(no_intervals - install_int)* plus the number of years left multiplied by *no_interval*s). This is then set to the return value *temp_time*. If the maintenance event will be due before the project ends (i.e. *maint_due_year* is not zero), however, then the *install_int* value is used to calculate the number of intervals from then until the maintenance event is due (i.e. *(maint_due_interval - install_int) + (no_intervals * (maint_due_year - temp_year))*), and is stored as *temp_time*. Before the next WEC-based maintenance event is considered, the custom function *min* (section 7.1.7) is used with the arguments *prev_time* and *temp_time*, and the minimum value then becomes the new *temp_time*. After all maintenance events have been considered, the function name (*ints_to_next_maint*) is set to the value of the *temp_time* for use by the calling procedures.

7.14.33 Number of retrieval failures

The function *num_retrieval_fails* is used to calculate the number of failures in a given list (*fail_arr*) which require the action "Retrieve WEC". It is called by the procedure *assign_lost_revenue_fails_maint* in the *array_object* class module (section 7.13.8). The function loops through each of the failures in *fail_arr* and uses the returned value from *get_action_reqd* in the relevant *fail_param_list* object (section 7.3) to identify if the failure needs the WEC to be retrieved (i.e. "Retrieve WEC"). If so then the return value (*temp_ret*, previously initialised as zero) is updated by adding 1. The function name (*num_retrieval_fails*) is set to be *temp_ret* for use by the calling procedures.

7.14.34 This maintenance ready

WEC-based maintenance events are defined in the *start* subroutine of the *wec_object* class module (section 7.14.1). As described, the IDs of these events are stored in the variable array *wec_maint_cat*, with the arrays *maint_due* and *maint_checker* controlling when the tasks are to be carried out. The variable *set_for_maint* has the data type *yes_no* (section 7.1.1) and allows the model to control the user-defined amount of space that can be used solely for scheduled maintenance activities (specified on the 'Inputs' spreadsheet, section 4.1.1). Other procedures and class modules can utilise the following five functions in the *wec_object* to access information about the status of one or all WEC-based maintenance categories:

- *this_maint_ready* - Boolean, True if a given event is ready to be undertaken
- *any_maint_ready* - Boolean, True if any events are ready to be undertaken
- *get_num_wec_maints_ready* - Integer, the number of events ready to be undertaken
- *any_maint_due* - Boolean, True if any events are due
- *get_num_wec_maints_due* - Integer, the number of events due

The Boolean function *this_maint_ready* is used throughout the model code to identify if a given WEC-based maintenance event (*this_cat*) is scheduled for maintenance at the current interval and if there is enough space at the O&M base for another WEC to undergo maintenance. Error handling is included in the function to ensure that the given category (*this_cat*) is relevant to the WEC. The

Boolean value of *this_maint_ready* is returned to the calling procedure as True if the relevant entry in *maint_due* is 1 (i.e. saying that the event is due) and the value of *set_for_maint* is *yes*. If one or both of these conditions is not met then the function returns False.

The Boolean function *any_maint_ready* is used to find if any of the WEC-based maintenance events are ready (or are being) carried out. It achieves this by looping through each entry in *wec_maint_cat* and changing the return value to True if the value of the *this_maint_ready* function for that event is True. If no maintenance events ready to be undertaken are found then the *any_maint_ready* function returns False.

The function *get_num_wec_maints_ready* operates in very much the same way as *any_maint_ready* described in the previous paragraph. The difference is that the return value (*temp_int*) is initialised as zero and then added to for every category returned as True by *this_maint_ready*.

The function *any_maint_due* is used to identify whether any WEC-based maintenance events are due to be undertaken, without considering the amount of space available at the O&M base. It achieves this by looping through all the WEC-based maintenance events and changing the return value to True if any of them have the relevant entry in *maint_due* set as 1. If all the maintenance events have *maint_due* set as zero then the return value remains as False.

The function *get_num_wec_maints_due* operates in very much the same way as *any_maint_due* described in the previous paragraph. The difference is that the return value (*temp_int*) is initialised as zero and then added to for every category whose *maint_due* entry is 1.

7.14.35 Any maintenance delayed

The Boolean function *any_maint_delay* is used to identify if any WEC-based maintenance events are being delayed due to a lack of space at the O&M base specifically for WECs undergoing maintenance only. The return value *temp_bool* is first initialised to False, before each WEC-based maintenance event is considered in a *for* loop with the identifier *i*. If the event is due (i.e. *if maint_due(i) = 1*) but the space at the O&M base restricts the work (i.e. *if set_for_maint = no*) then *temp_bool* is set to True. After the loop, the function name *any_maint_delay* is set to *temp_bool* for use by the calling procedure.

7.14.36 Get functions

There are a number of functions throughout the *wec_object* class module which are used by other objects purely to obtain information about that specific WEC. The names of these functions all start with '*get_*' and are as follows:

- *get_state* - returns the *state* of the WEC
- *get_maint_due* - obtains the *maint_due* entry of a given *this_entry*
- *get_wec_power* - gets *wec_power* (after checking rounding errors)
- *get_fail_list* - gets the *wec_fail_list* object (known as *wec_fail_arr*)
- *get_wec_output_list* - gets the *wec_output_list* object (known as *wec_output_arr*)
- *get_delay_status* - gets the *delay_status*
- *get_wec_maint_cat* - gets the *wec_maint_cat* entry of a given *this_entry*
- *get_num_wec_maint_cats* - gets the upper boundary of *wec_maint_cat*

7.15 TECHNICIANS

The *technicians_object* class module is created by the *array_object* (and known as *technicians*) in order to set up and control the workforce arrangements at the O&M base. A number of variables are used throughout the object:

- *tech_available()* - Boolean, contains a list of technicians' availability
- *annual_labour_cost* - annual labour cost of O&M base technicians
- *techs_work_time_remaining()* – a list of the number of intervals each technician has left on a task
- *techs_output_arr* - a list of *techs_output* objects
- *contractor_day_rate* - daily hire rate for an external contractor
- *contractor_int_rate* - hire rate for an external contractor per interval
- *contractors_on_hire()* - Boolean, whether contractors are on hire in each interval and year

In addition, the constant values (*Const*) *totals_row* and *overheads_col* are used to identify the position in the 'Labour' spreadsheet (section 4.3) of the row containing the totals of personnel and salary and the column containing the overheads multiplier respectively.

7.15.1 Start

The subroutine *start* is called by the procedure of the same name in the *array_object* class module (section 7.13.1) in order to initialise the *technicians* object. The value of the *annual_labour_cost* is calculated by the product of total salaries of the O&M base technicians and the overheads multiplier, both specified on the 'Labour' spreadsheet (section 4.3). The *contractor_day_rate* is read from the relevant cell and converted into a fee per interval (*contractor_int_rate*) by dividing by the number of intervals in a day (i.e. 24 / *time_step*). It should be noted that if the *labour_shee*t is modified then the cells to read the information from here must also be changed accordingly.

The number of technicians at the O&M base (*num_technicians*) was identified during the procedure *setup_class* (see section 7.2.4) for use throughout the model. It is used here to re-size the arrays *tech_available* and *techs_work_time_remaining* from 1 to *num_technicians*, so that information about an individual O&M base technician can be accessed. A *for* loop then considers each technician with the identifier *i* and the relevant entries in these arrays are initialised to say that each technician is available and not undertaking work (i.e. *tech_available(i) = True* and *techs_work_time_remaining(i) = 0*).

The *Global* Boolean variable *short_term_contracts_enabled* is then initialised to False, saying that external contractors cannot be hired. This is only changed to True if the relevant cell in the 'Labour' spreadsheet (section 4.3) reads "yes". The variable *contractors_on_hire* is then re-sized to be a 2D array, with the first dimension providing space for information about every interval in a year (i.e. from *1 to no_intervals*) and the second dimension for every year in the project lifetime (i.e. from *1 to no_run*). A nested loop then considers each interval (identifier *i*) in each year (identifier *irun*) and initialises the relevant entry in *contractors_on_hire* (i.e. *contractors_on_hire(i, irun)*) to False.

Finally, the subroutine creates and initialises an output object to keep track of the contractor fees incurred in each year of the project lifetime, as well as a zero entry to contain averaged data. The *techs_output_arr* variable is re-sized as a 2D array (i.e. *techs_output_arr(0 To no_run, 1 To 1)*) before a nested *for* loop considers each entry in turn. A *New techs_output* object is assigned to the relevant entry of *techs_output_arr* and its *start* subroutine is called (section 7.21). It is clearly

unnecessary to have *techs_output_arr* as a 2D array when the second dimension only has one entry. However, it has been included here in order to provide the framework to output information about every O&M base technician if required by future users of the model.

7.15.2 Add technicians working

The subroutine *add_tech_working* is used to update the two arrays *tech_available* and *techs_work_time_remaining* for a given O&M base technician (*this_technician*). It is called by the *array_object* subroutine *attempt_fix* (section 7.13.4) and the *wec_object* procedures *attempt_fix (section* 7.14.4), *next_interval* (section 7.14.14), *assign_offsite_fail_techs* and *assign_offsite_maint_techs* (section 7.14.18). The subroutine should only be called when *this_technician* has already been identified as available for work, so error handling displays a message if the relevant value of *tech_available* is False. Otherwise, the entry is set to be False, saying that the technician is now busy. An available technician should also have the relevant entry in *techs_work_time_remaining* set to zero, otherwise an error message will be displayed. If the value in *techs_work_time_remaining* is zero, then it is reset to be the number of *working_intervals* sent by the calling procedure.

7.15.3 Add contractor fees

The subroutine *add_contractor_fees* is used to update the output information whenever an external contractor is required for marine operations, repairs or maintenance tasks. It is called by the *array_ object* subroutine *attempt_fix* (section 7.13.4) and the *wec_object* procedures *attempt_fix (section* 7.14.4), *next_interval* (section 7.14.14), *assign_offsite_fail_techs* and *assign_offsite_maint_techs* (section 7.14.18). The subroutine is sent date information in the form *irun* (the current year) and *this_interval* (current interval in *irun*). It is also sent the number of intervals that contractors will be hired for (*num_ints*) as well as an identifier *start_next* (with the data type *yes_no*). The *add_contractor_fees* subroutine acts as a control by first calling another *technicians* subroutine, *update_contractors_on_hire*, with all the arguments described. It then calls the *add_contractor_ fees* procedure in the *techs_output_arr* object relevant to *irun* (section 7.21.5) using the argument of *contractor_int_rate* multiplied by *num_ints*.

The subroutine *update_contractors_on_hire* is used to change the relevant entries of the *contractors_on_hire* array to True whenever external contractors will be used. If the calling procedure of *add_contractor_fees* is the *wec* subroutine *next_interval*, then the identifier *start_next* is set as *yes*, saying that the contractors will be on hire starting at the next interval. This is accounted for by making the final interval (*end_int*) in every other case equal to *this_interval* plus *num_ints* minus 1. The minus 1 is not needed when *next_interval* is the calling procedure. The subroutine then considers every interval from *this_interval* to *end_int* in a *for* loop with the identifier *i*. It is possible that the interval under consideration will go into the next year (i.e. *irun* plus 1). If that is not the case then the variables *array_interval* and *array_year* are set to *i* and *irun* respectively. If the interval does go into the next year, however, then *array_interval* becomes *i* minus *no_intervals* (the number of intervals in a year) and *array_year* is set to *irun* plus 1. In the rare case that the considered interval goes beyond the project lifetime, *array_interval* is set to *no_intervals* and *array_year* becomes *irun* (i.e. the last interval of the project). Before the next interval is considered or the loop ends, the relevant entry in *contractors_on_hire* (i.e. *contractors_on_hire(array_interval, array_year)*) is set to True.

7.15.4 Next interval

The subroutine *next_interval* is called by the procedure of the same name in the *array_object* class module (section 7.13.7). It is used to set the *technicians* object up for the coming interval.

A *for* loop considers each O&M base technician in turn (i.e. from *1 to num_technicians*) with the identifier *i*. The relevant entry in the *techs_work_time_remaining* array is updated by subtracting one, saying that the technician has worked one more interval (if busy). The value in *techs_work_time_remaining* is set to zero if this makes it become a negative number. If the technician under consideration is not working at the current interval (i.e. *techs_work_time_remaining(i) = 0* and *tech_available(i) = False*) then the relevant entry in the Boolean array *tech_available* is changed to True, making that technician available for work once more.

7.15.5 Print interval

The subroutine *print_interval* is called by the procedure of the same name in the *array_object* class module (section 7.13.12) in order to print output information about the technicians to the 'run sheets' (section 6.2) is a 'full run' process is taking place (section 5.2). It is sent the name of the output sheet (*run_sh*), the current year (*irun*), the current interval (*this_interval*) and the first column to start printing (*techs_start_col*).

If *this_interval* is the first interval in the year then the section header "Technicians" is printed. A *for* loop then considers each of the O&M base technicians (i.e. from *1 to num_technicians*) with the identifier *i*. Again, if *this_interval* is 1 then the sub-section header is printed in the correct column (i.e. *techs_start_col + i – 1*), indicating the ID of that technician (*i*). If the technician under consideration is available at that interval (i.e. *if tech_available(i) = True*) then the relevant cell is made to read "Free" and is filled green. Otherwise, it reads "Busy" and is filled red. Following the O&M base technicians loop, the contractor information is printed. The section header reads "Contractors on hire?" in the last column (*techs_start_col + num_technicians*). If the relevant value of *contractors_on_hire* for the current date is True then the cell reads "yes" and is filled red. Otherwise, it reads "no" and is filled green.

7.15.6 Get functions

Information from the *technicians* object can be obtained by other class modules using the following functions:

- *get_tech_availability* - the value of the *tech_available* array given *this_tech*
- *get_annual_labour_cost* - the *annual_labour_cost* calculated in *start*
- *get_num_techs_avail* - loops through the O&M base technicians and calculates how many of them are available
- *get_techs_output* - the *techs_output_arr* object given the first dimension *i*

7.15.7 Output procedures

The procedures *draw, run_title, calc_end, post_process_contractor_fees* are used to produce the technicians-related output information printed to the 'Results' spreadsheet (section 6.1). They are described in detail in section 7.21.

7.16 FAILURES OBJECTS

The class modules *array_fail_list* and *wec_fail_list* control the list of array-based and wec-based failures respectively which have been simulated at any given interval. As has been described throughout sections 7.13 and 7.14, the procedures in these objects are called in order to update and obtain the list of sustained failures. The objects utilise the class modules *array_fail* and *wec_fail* respectively for each of the suffered failures. The class module *failure_no_object* is also used in order to identify the classifications of the sustained faults. Some of these failure objects have common variables:

- *fail_no* - Integer, number of failures in the list
- *fail_arr()* - List of *array_fail* or *wec_fail* objects
- *fail_number* - A *failure_no_object* class module
- *fail_ID* - ID of fault category

7.16.1 Array failures list

The class module *array_fail_list* is used throughout the *array_object* and is referred to as *array_fail_arr* (see section 7.13). It controls the list of array-based failures sustained.

The subroutine *start* is called by the procedure of the same name in *array_object* (section 7.13.1), as well as the *next_interval* procedure (section 7.13.7), in order to reset the list. The total number of failures, *fail_no*, is set to zero. The object controlling the classification information of the failures is then created and initialised by setting *fail_number* to be a *New fail_no_object* and calling its *start* procedure (section 7.16.5).

The subroutine *add_fail* is called during the *determine_failure* procedure in *array_object* (section 7.13.2) in order to update the list with the sustained failure. The ID of the failure is sent as *fail_type*. The number of sustained failures, *fail_no*, is updated by adding 1. The list of failures, *fail_arr*, is then re-sized from 1 to the current value of *fail_no*, but keeping the existing values (i.e. *ReDim Preserve*). The new entry of the list (i.e. *fail_arr(fail_no)*) is set to be a *New array_fail* object and the ID of the failure (*fail_type*) is sent to its *start* procedure (section 7.16.2). The severity of the sustained failure is then read using the *get_severity* function in the relevant object (i.e. *fail_param_list.get_fail_param(fail_type)*, section 7.3) and is stored in the variable *sev* (with the data type *severity*, see section 7.1.1). The *fail_number* object is then updated by sending the value of *sev* to its *count* procedure (section 7.16.5).

The function *get_fail_arr_id* is used to return a list of all the array-based failures which have been sustained. It is called by several procedures throughout the *array_object* class module (see section 7.13) whenever the list of failures is required. If there are no failures (i.e. if *fail_no* is zero) then the first and only entry of the return value (i.e. *arr(1)*) is set to be -5. Otherwise, the return list is re-sized from 1 to the value of *fail_no*, and a *for* loop fills each entry (i.e. *arr(i)*) with the ID of that failure (i.e. *fail_arr(i).get_fail_id*, section 7.16.2). The function name, *get_fail_arr_id*, is set to be the value of *arr* so it can be recognised by the calling procedures.

The function *string_fail* is called by the *print_interval* subroutine in the *array_object* class module (section 7.13.12). It is used to create a String list (i.e. text) of all the sustained array-based failures. The String return value (*string_fail_loc*) is first initialised to say “fail :”. If *fail_no* is greater than zero then each sustained failure is considered in turn using a *for* loop (from 1 to *fail_no*). The value of *string_fail_loc* is updated each time by adding the ID of that failure (i.e. *fail_arr(i).get_fail_id*,

section 7.16.2). Formatting is taken into account by adding a comma and a space unless the last failure in the list is being considered. The function name, *string_fail*, is set to be the value of *string_fail_loc* so it can be recognised by the calling procedure.

Procedures in other class modules can access the total number of sustained array-based failures (*fail_no*) by calling the function *get_total_fails*. They can also access the object containing the severity of the sustained failures (*fail_number*) using the function *get_fail_number*.

7.16.2 Array failures

The *array_fail* class module is used by the *array_fail_list* object (section 7.16.1) to store the ID of an array-based failure which has been sustained. The *start* subroutine takes the argument of *fail_type* and assigns it to the variable *fail_ID*. The function *get_fail_id* can then be called by the *array_fail_list* object in order to access the value of *fail_ID.*

7.16.3 WEC failures list

The class module *wec_fail_list* is used throughout the *wec_object* and is referred to as *wec_fail_arr* (see section 7.14). It controls the list of wec-based failures sustained.

The subroutine *start* is called by the procedure of the same name in *wec_object* (section 7.14.1), as well as the *next_interval* procedure (section 7.14.14), in order to reset the list. The total number of failures, *fail_no*, is set to zero. The object controlling the classification information of the failures is then created and initialised by setting *fail_number* to be a *New fail_no_object* and calling its *start* procedure (section 7.16.5).

The subroutine *add_fail* is called during the *determine_failure* procedure in *wec_object* (section 7.14.2) in order to update the list with the sustained failure. The ID of the failure is sent as *fail_type*. The number of sustained failures, *fail_no*, is updated by adding 1. The list of failures, *fail_arr*, is then re-sized from 1 to the current value of *fail_no*, but keeping the existing values (i.e. *ReDim Preserve*). The new entry of the list (i.e. *fail_arr(fail_no)*) is set to be a *New wec_fail* object and the ID of the failure (*fail_type*) is sent to its *start* procedure (section 7.16.4). The severity of the sustained failure is then read using the *get_severity* function in the relevant object (i.e. *fail_param_list.get_fail_param(fail_type)*, section 7.3) and is stored in the variable *sev* (with the data type *severity*, see section 7.1.1). The *fail_number* object is then updated by sending the value of *sev* to its *count* procedure (section 7.16.5).

The function *get_fail_arr_id* is used to return a list of all the wec-based failures which have been sustained. It is called by several procedures throughout the *wec_object* class module (see section 7.14) whenever the list of failures is required. If there are no failures (i.e. if *fail_no* is zero) then the first and only entry of the return value (i.e. *arr(1)*) is set to be -5. Otherwise, the return list is re-sized from 1 to the value of *fail_no*, and a *for* loop fills each entry (i.e. *arr(i)*) with the ID of that failure (i.e. *fail_arr(i).get_fail_id*, section 7.16.4). The function name, *get_fail_arr_id*, is set to be the value of *arr* so it can be recognised by the calling procedures.

The function *string_fail* is called by the *print_interval* subroutine in the *wec_object* class module (section 7.14.20). It is used to create a String list (i.e. text) of all the sustained wec-based failures. The String return value (*string_fail_loc*) is first initialised to say “fail :”. If *fail_no* is greater than zero then each sustained failure is considered in turn using a *for* loop (from 1 to *fail_no*). The value of *string_fail_loc* is updated each time by adding the ID of that failure (i.e. *fail_arr(i).get_fail_id*,

section 7.16.4). Formatting is taken into account by adding a comma and a space unless the last failure in the list is being considered. The function name, *string_fail*, is set to be the value of *string_fail_loc* so it can be recognised by the calling procedure.

Procedures in other class modules can access the total number of sustained wec-based failures (*fail_no*) by calling the function *get_total_fails*. They can also access the object containing the severity of the sustained failures (*fail_number*) using the function *get_fail_number*.

The subroutine *update_fail_arr* is called by the *wec* procedure *next_interval* (section 7.14.14) when onsite repairs have taken place. As described throughout section 7.14, the model assumes that only one part can be replaced onsite during a single marine operation. Therefore, it is possible that the list of sustained wec-based failures will only be partially cleared. The calling procedure sends *update_fail_arr* a list of the failures which have been corrected (*fails_to_remove*). The subroutine starts by storing the list of all sustained failures in the variable *current_fail_arr*. It then clears the list by calling the *start* subroutine described previously in this section. Each failure sustained by the failure is then considered in a *for* loop from 1 to the number of entries in *current_fail_arr* with the identifier *i*. The ID failure under consideration (*current_fail*) is obtained from the list (i.e. *current_fail_arr(i)*) and a counter (*count*) is initialised to zero. A nested *for* loop then considers each entry in the *fails_to_remove* list using the identifier *k*. If the failure under consideration (*current_fail*) is in the list (i.e. *if fails_to_remove(k) = current_fail*) then the value of *count* is updated by adding 1. If *count* is still zero after all entries of *fails_to_remove* have been assessed then the *current_fail* is once again added to the list of sustained wec-failures by calling the subroutine *add_fail*.

7.16.4 WEC failures

The *wec_fail* class module is used by the *wec_fail_list* object (section 7.16.3) to store the ID of an wec-based failure which has been sustained. The *start* subroutine takes the argument of *fail_type* and assigns it to the variable *fail_ID*. The function *get_fail_id* can then be called by the *wec_fail_list* object in order to access the value of *fail_ID.*

7.16.5 Failure number object

The class module *fail_no_object* is used to store classification information about the sustained array-based and wec-based failures. It stores the number of occurrences of major (*major_count*), intermediate (*intermediate_count*) and minor failures (*minor_count*). The *start* subroutine initialises each of these counters by setting them to zero.

The subroutine *count* is sent the severity of a failure in the variable *sev*, with the data type *severity* (section 7.1.1). An *if-else* set of conditions then identifies whether *sev* is *major, intermediate* or *minor* and updates the relevant counter appropriately by adding 1.The values of the counters can be accessed at any time by using the functions *ret_major_count*, *ret_intermediate_count* and *ret_minor_count*.

7.17 ARRAY OUTPUTS LIST

The class module *array_output_list* is used to update, control and calculate the outputs related to the array as a whole. The following variable names are used throughout the object:

- *num_wecs* - the number of WECs in the array
- *array_output()* - a list of *wec_output* objects used to store array outputs
- *all_wecs_output()* - a list of *wec_output_list* objects used to calculate outputs from all WECs
- *total_parts_costs()* - a list of the total parts costs incurred in each year of the project
- *total_other_costs()* - a list of the total other costs incurred in each year of the project
- *total_inspection_costs()* - a list of the total inspection costs incurred in each year of the project

7.17.1 Start

The subroutine *start* is called by the procedure of the same name in the *array_object* class module (section 7.13.1) in order to setup and initialise the array-based output objects for each year of the project lifetime. It is sent the number of WECs in the array (*num_wecs_loc*) as an argument which is then set to be the value of *num_wecs*.

The list of output objects, *array_output*, is then re-sized from zero to the number of years in the project lifetime (*no_run*). The zero entry (i.e. *array_output(0)*) is used to store the average values across the entire project lifetime. Each entry is then considered in a *for* loop using the identifier *i*. Each entry (i.e. *array_output(i)*) is then set to be a *New wec_output* object and its subroutine *start* is called (section 7.19.1).

Note: the class module *wec_output* is used to store output information for array-based and wec-based aspects. The calling procedure determines which aspect is under consideration.

7.17.2 Add failure and maintenance costs

The subroutines *fail_costs* and *add_maint_costs* are called by the *next_interval* procedure in the *array_object* (section 7.13.7) in order to update the outputs costs for repaired failures and completed maintenance tasks respectively.

The *fail_costs* subroutine is sent the current year of the simulation (*irun*) and a list of the IDs of repaired failures (*fail_arr_id*). It then sends *fail_arr_id* to the *costs_add* subroutine of the correct *array_output* object for that value of *irun* in order to update the incurred costs (section 7.19.3).

The *add_maint_costs* subroutine is sent the current year of the simulation (*irun*) and the ID of the completed array-based maintenance task (*maint_cat*). It then sends *maint_cat* to the *maint_costs_add* subroutine of the correct *array_output* object for that value of *irun* in order to update the incurred costs (section 7.19.3). A list of maintenance tasks is not required here because only one array-based maintenance task is coded for in the released version of the O&M mode, as described in section 7.13.1.

7.17.3 Add availability

The subroutine *avail_add* is called by the *next_interval* procedure in the *array_object* (section 7.13.7) in order to update the output information for wave farm availability. It is sent the current

year of the simulation (*irun*) and the current power capacity of the array (*power*, between 0 and 1). The subroutine updates the average power incurred in that year by using the *set_avail* subroutine of the correct *array_output* object for that value of *irun* (section 7.19.2). The new average availability is calculated by obtaining the previous average availability (*get_avail*) and adding the current *power* capacity divided by the number of intervals in a year (*no_intervals*).

7.17.4 Draw

The *draw* subroutine is called at the end of a simulated array lifetime by the *post_process* procedure in the *array_object* class module (section 7.13.13) in order to control the printing of array-based outputs to the 'Results' spreadsheet (section 6.1). The subroutine is sent the reference IDs of the *row* and column (*col*) positions to start printing the information. The annual average output values are calculated by calling the subroutine *calc_end* (section 7.17.5). The output information for each year of the project lifetime is then printed by calling the following procedures in order, updating the *row* to start printing each time (section 7.17.6):

- *post_process_avail*
- *post_process_part_cost*
- *post_process_other_cost*
- *post_process_inspection_cost*

In addition, the *draw* subroutine also prints the total costs incurred in every year, including array-based aspects as well as all wec-based aspects. This is achieved by calling the subroutines *set_total_parts_costs, set_total_other_costs* and *set_total_inspection_costs* (section 7.17.8).

7.17.5 Calculate end

The subroutine *calc_end* is called by *draw* (section 7.17.4) in order to fill the zero entry of the *array_output* list (i.e. *array_output(0)*) with the values of array availability, parts costs, other costs and inspection costs, averaged across the lifetime of the project. Each year is considered in turn using a *for* loop with the identifier *i*, from 1 to *no_run.* For each year, the relevant 'set' subroutines (e.g. *set_avail* etc.) in the *array_output(0)* object (section 7.19.2) are called. The argument sent in each case is the current average value (e.g. *get_avail*) plus the value for the year under consideration (e.g. *array_output(i).get_avail*) divided by *no_run*.

7.17.6 Post process procedures

The four 'post process' subroutines in *array_output_list* print the annual array-based output information for availability (*post_process_avail*), parts costs (*post_process_part_cost*), other costs (*post_process_other_cost*) and inspection costs (*post_process_inspection_cost*). In each case, the header "Array" is printed in the appropriate cell. Each year of the project lifetime is considered in a *for* loop with the identifier *i*. The output information for that year is then obtained from the relevant *array_output* object (i.e. *array_output(i)*, section 7.19.2) and printed to the appropriate cell (i.e. *col + i*). In the cost-based subroutines, the *output_val_divider* is used to convert the information into the required format defined by the user (section 7.2.4). Once the information is printed for every year, the annual average values calculated in *calc_end* (section 7.17.5) are also printed to the 'Results' spreadsheet.

7.17.7 Draw all WECs

The subroutine *draw_all_wecs* is called at the end of a simulated array lifetime by the *post_process* procedure in the *array_object* class module (section 7.13.13). It is called after the output information for each WEC has been printed (section 7.18). It is used to print the sum of the output information for all WECs in the array to the 'Results' spreadsheet (section 6.1). The subroutine is sent the reference IDs of the *row* and column (*col*) positions to start printing the information, as well as the output object for each WEC (stored in a list defined as *wec_output_arr_loc*). The list *wec_output_arr_loc* is then stored as *all_wecs_output*.

Each year of the project lifetime is considered in a *for* loop with the identifier *i_year*, as well as the zero entry of the output objects (i.e. from 0 to *no_run*). The header "All wecs" is printed for each of the four output aspects (availability, parts costs, other costs and inspection costs). The rows where these headers are printed are calculated using the *num_wecs* and must be modified if the layout of the 'Results' spreadsheet is changed. The headers are printed if the value of *i_year* is zero, thereby printed them only once.

The average availability from all WECs for the year under consideration (*ave_avail*) is initialised to zero. The sums of the parts, other and inspection costs of all WECs for the year under consideration (*sum_parts*, *sum_other* and *sum_inspection* respectively) are also initialised to zero. A nested *for* loop then considers each WEC in the array (i.e. *i_wec* from 1 to *num_wecs*). The average availability is updated by adding the value of *get_avail* obtained from the relevant WEC output object (i.e. *all_wecs_output(i_wec).get_wec_output(i_year)*) divided by *num_wecs*. The same object is used for the functions *get_part_cost*, *get_other_cost* and *get_inspection_cost* in order to update the sum values of *sum_parts*, *sum_other* and *sum_inspection* respectively. Following the nested WECs loop, these values are then printed to the correct cell in the 'Results' spreadsheet.

7.17.8 Set total costs

The total costs incurred by the project (i.e. including array-based aspects as well as all wec-based aspects) are set by the subroutines *set_total_parts_costs, set_total_other_costs* and *set_total_inspection_costs*. These are called by the *draw* procedure (section 7.17.4). In each subroutine, the list of total costs (i.e. *total_parts_costs, total_other_costs* or *total_inspection_costs*) is re-sized from zero to the number of years of the project lifetime (*no_run*). The header "Total" is printed in the relevant cell. Each entry in the list of costs is then considered in a *for* loop where the sum of the values from all WECs (defined during *draw_all_wecs*, section 7.17.7) are added to the array-based outputs and printed in the relevant cell.

The lists of total costs can be accessed using the functions *get_total_parts_costs, get_total_other_costs* and *get_total_inspection_costs*, with *i* being the entry identifier.

7.17.9 Get functions

Two other functions in *array_output_list* can be utilised by procedures in other objects. The function *get_array_output* is sent an identifier *i* and returns the specified *array_output* object. The function *get_no_param* obtains the number of parameters printed to the 'Results' spreadsheet by using the *array_output(0)* object (section 7.19.4).

7.18 WEC OUTPUT LIST

The class module *wec_output_list* is used to update, control and calculate the outputs related to a particular WEC in the wave energy array. The following variable names are used throughout the object:

- *wec_id* - the ID of the WEC
- *name* - header
- *wec_output_arr()* - a list of *wec_output* objects used to store WEC outputs

7.18.1 Start

The subroutine *start* is called by the procedure of the same name in the *wec_object* class module (section 7.14.1) in order to setup and initialise the WEC-based output objects for each year of the project lifetime. It is sent the ID of WEC (*id*) which is subsequently set to the value of *wec_id*. The variable *name* defines the header to be printed to the *results_sheet* and is set to be "WEC " plus the *wec_id*.

The list of output objects, *wec_output_arr*, is then re-sized from zero to the number of years in the project lifetime (*no_run*). The zero entry (i.e. *wec_output_arr(0)*) is used to store the average values across the entire project lifetime. Each entry is then considered in a *for* loop using the identifier *i*. Each entry (i.e. *wec_output_arr(i)*) is then set to be a *New wec_output* object and its subroutine *start* is called (section 7.19.1).

7.18.2 Add failure and maintenance costs

The subroutines *fail_costs* and *add_maint_costs* are called by the *next_interval* procedure in the *wec_object* (section 7.14.14) in order to update the outputs costs for repaired failures and completed maintenance tasks respectively.

The *fail_costs* subroutine is sent the current year of the simulation (*irun*) and a list of the IDs of repaired failures (*fail_arr_id*). It then sends *fail_arr_id* to the *costs_add* subroutine of the correct *wec_output_arr* object for that value of *irun* in order to update the incurred costs (section 7.19.3).

The *add_maint_costs* subroutine is sent the current year of the simulation (*irun*) and the ID of the completed WEC-based maintenance task (*maint_cat*). It then sends *maint_cat* to the *maint_costs_add* subroutine of the correct *wec_output_arr* object for that value of *irun* in order to update the incurred costs (section 7.19.3). A list of tasks is not required here because a *for* loop in *next_interval* calls this subroutine for each completed WEC-based maintenance event, as described in section 7.14.14.

7.18.3 Add availability

The subroutine *avail_add* is called by the *next_interval* procedure in the *wec_object* (section 7.14.14) in order to update the output information for WEC availability. It is sent the current year of the simulation (*irun*) and the current power capacity of the WEC (*power*, between 0 and 1). The subroutine updates the average power incurred in that year by using the *set_avail* subroutine of the correct *wec_output_arr* object for that value of *irun* (section 7.19.2). The new average availability is calculated by obtaining the previous average availability (*get_avail*) and adding the current *power* capacity divided by the number of intervals in a year (*no_intervals*).

7.18.4 Draw

The *draw* subroutine is called at the end of a simulated array lifetime by the *post_process* procedure in the *array_object* class module (section 7.13.13) in order to control the printing of WEC-based outputs to the 'Results' spreadsheet (section 6.1). The subroutine is sent the reference IDs of the *row* and column (*col*) positions to start printing the information, as well as the value of *num_wecs* (the number of WECs in the array) to assist with printing. The annual average output values are calculated by calling the subroutine *calc_end* (section 7.18.5). The output information for each year of the project lifetime is then printed by calling the following procedures in order, updating the *row* to start printing each time (section 7.18.6):

- *post_process_avail*
- *post_process_part_cost*
- *post_process_other_cost*
- *post_process_inspection_cost*

7.18.5 Calculate end

The subroutine *calc_end* is called by *draw* (section 7.18.4) in order to fill the zero entry of the *wec_output_arr* list (i.e. *wec_output_arr(0)*) with the values of WEC availability, parts costs, other costs and inspection costs, averaged across the lifetime of the project. Each year is considered in turn using a *for* loop with the identifier *i*, from 1 to *no_run.* For each year, the relevant 'set' subroutines (e.g. *set_avail* etc.) in the *wec_output_arr(0)* object (section 7.19.2) are called. The argument sent in each case is the current average value (e.g. *get_avail*) plus the value for the year under consideration (e.g. *wec_output_arr(i).get_avail*) divided by *no_run*.

7.18.6 Post process procedures

The four 'post process' subroutines in *wec_output_list* print the annual WEC-based output information for availability (*post_process_avail*), parts costs (*post_process_part_cost*), other costs (*post_process_other_cost*) and inspection costs (*post_process_inspection_cost*). In each case, the subroutine *run_title* (section 7.18.7) is called if *wec_id* is 1 (i.e. so the header is only printed once) with the required header (e.g. "Availability"). The header *name* (e.g. "WEC 1") is also printed in the appropriate cell. Each year of the project lifetime is considered in a *for* loop with the identifier *i*. The output information for that year is then obtained from the relevant *wec_output_arr* object (i.e. *wec_output_arr(i)*, section 7.19.2) and printed to the appropriate cell (i.e. *row + wec_id* and *col + i + 1*). In the cost-based subroutines, the *output_val_divider* is used to convert the information into the required format defined by the user (section 7.2.4). Once the information is printed for every year, the annual average values calculated in *calc_end* (section 7.18.5) are also printed to the 'Results' spreadsheet.

7.18.7 Run title

The subroutine *run_title* is called by the 'post process' procedures (section 7.18.6) in order to print the specified header (*this_parameter*) to the appropriate cell in the 'Results' spreadsheet (section 6.1). If the parameter is cost-based then the variable *output_money_format* (section 7.2.4) is printed alongside *this_parameter*. Only *this_parameter* is printed in the case of "Availability". The subroutine also prints the year headers (i.e. "year " & *i*), as well as "per year", in the appropriate cells.

7.18.8 Get functions

Two other functions in *wec_output_list* can be utilised by procedures in other objects. The function *get_wec_output* is sent an identifier *i* and returns the specified *wec_output_arr* object. The function *get_no_param* obtains the number of parameters printed to the 'Results' spreadsheet by using the *wec_output_arr(0)* object (section 7.19.4).

7.19 WEC OUTPUT

The *wec_output* object is used to store output information of the four key parameters (availability, parts costs, other costs and inspection costs) for either the array or for each WEC. As described in sections 7.17 and 7.18, the relevance of *wec_output* (i.e. whether it is for the array or for each WEC) is determined by the calling procedure (i.e. *array_output_list* or *wec_output_list*). The parameters are defined using the variables *avail, part_cost, other_cost* and *inspection_cost*. The number of parameters (*no_param*) is a constant value (*Const*) set to 4.

7.19.1 Start

The *start* subroutine is called by the procedures of the same name in the *array_output_list* (section 7.17.1) and *wec_output_list* class modules (section 7.18.1) in order to initialise the outputs. The values of *avail, part_cost, other_cost* and *inspection_cost* are all set to zero.

7.19.2 Set and get values

Each of the parameters has a 'set' subroutine and a 'get' function (e.g. *set_avail* and *get_avail*). The 'set' subroutines are sent a value to set to the relevant parameter (e.g. *avail*). This value is calculated using the 'get' functions, as described throughout sections 7.17 and 7.18.

7.19.3 Add costs

The subroutine *add_costs* is called by the *fail_costs* procedures in the *array_output_list* and *wec_output_list* objects (sections 7.17.2 and 7.18.2 respectively) in order to update the costs incurred after completing a list of repairs. A list of the IDs of the fault category which have been repaired is sent in the variable *fail_arr*. If there are no failures then the first and only entry of *fail_arr* will be negative. Otherwise, each failure is considered in turn using a *for* loop with the identifier *i*. For each failure, the functions *get_part* and *get_other* in the relevant *fail_param_list* object (i.e. *fail_param_list.get_fail_param(fail_arr(i))*) are used to update the values of *part_cost* and *other_cost*.

The subroutine *maint_costs_add* is called by the *add_maint_costs* procedures in the *array_output_list* and *wec_output_list* objects (sections 7.17.2 and 7.18.2 respectively) in order to update the costs incurred after completing a particular maintenance task (*maint_cat*). The functions *get_part*, *get_other* and *get_inspection* in the relevant *maint_param_list* object (i.e. *maint_param_list.get_maint_param(maint_cat)*) are used to update the values of *part_cost*, *other_cost* and *inspection_cost*.

7.19.4 Number of parameters

The function *get_no_param* is used by other objects to obtain the value of *no_param*, in order to assist with the format of printing the outputs to the 'Results' spreadsheet.

7.20 REVENUE OUTPUT

The output information related to the revenue is controlled and printed by the *revenue_object* and the *revenue_output* class module. As described throughout section 7.7, the *revenue_object* not only acts as the control object for revenue-based aspects throughout the simulated project lifetime, it also controls the output information. In this regard, it acts as a similar object to *array_output_list* and *wec_output_list* described in sections 7.17 and 7.18 respectively. This includes creating and initialising a *revenue_output* object (known as *revenue_output_arr*) for each year in the array lifetime, as well as a zero entry to contain averaged information (section 7.7.1). The following procedures in the *revenue_ object* are used to control the output information and are described further in this section:

- *draw*
- *run_title*
- *calc_end*
- *post_process_earned_rev*
- *post_process_theory_rev*
- *post_process_lost_rev*

When the *revenue_output* objects are initialised by the *start* procedure (section 7.7.1), the three parameters of the total earned revenue (*sum_earned_rev*), the total revenue possible (*sum_theory_rev*) and the total lost revenue (*sum_lost_rev*) are initialised to zero.

7.20.1 Draw

The *draw* function is called at the end of a simulated array lifetime by the *post_process* procedure in the *maint_manager_object* class module (section 7.5.9) in order to control the printing of revenue-based outputs to the 'Results' spreadsheet (section 6.1). The function is sent the reference IDs of the *row* and column (*col*) positions to start printing the information. The headers for the years of the array lifetime are printed by calling the subroutine *run_title* (section 7.20.2). The annual average output values are calculated by calling the subroutine *calc_end* (section 7.20.3). The output information for each year of the project lifetime is then printed by calling the following procedures in order, updating the *row* to start printing each time (section 7.20.4):

- *post_process_earned_rev*
- *post_process_theory_rev*
- *post_process_lost_rev*

The function returns the ID of the new row to start printing by setting *draw* to be existing value of *row* plus the number of parameters involved in the revenue output (i.e. *revenue_output_arr(0).get_no_param*), as well as the required gap for formatting purposes.

7.20.2 Run title

The *run_title* subroutine is called by the *draw* function (section 7.20.1) in order to print the year headers, as well as "per year", in the appropriate cells in the *results_sheet*. Each year in the array lifetime (*no_run*) is considered in a *for* loop with the identifier *i*. The header is printed as "year " plus the value of *i*.

7.20.3 Calculate end

The subroutine *calc_end* is called by *draw* (section 7.20.1) in order to fill the zero entry of the *revenue_output_arr* list (i.e. *revenue_output_arr(0)*) with the values of earned revenue, theoretical revenue and lost revenue, averaged across the lifetime of the project. Each year is considered in turn using a *for* loop with the identifier *i*, from 1 to *no_run.* For each year, the relevant 'set' subroutines (e.g. *set_sum_earned_rev* etc.) in the *revenue_output_arr(0)* object (section 7.20.5) are called. The argument sent in each case is the current average value (e.g. *get_sum_earned_rev*) plus the value for the year under consideration (e.g. *revenue_output_arr(i). get_sum_earned_rev*) divided by *no_run*.

7.20.4 Post process procedures

The three 'post process' subroutines in the *revenue_object* print the annual output information for earned revenue (*post_process_earned_rev*), theoretical revenue if operating at full power (*post_process_theory_rev*) and lost revenue (*post_process_lost_rev*). The header *name* (e.g. "Sum earned") is also printed in the appropriate cell, along with the *output_money_format* (section 7.2.4). Each year of the project lifetime is considered in a *for* loop with the identifier *i*. The output information for that year is then obtained from the relevant *revenue_output_arr* object (i.e. *revenue_output_arr(i)*, section 7.20.5) and printed to the appropriate cell (i.e. *col + i*). The *output_val_divider* is used to convert the information into the required format defined by the user (section 7.2.4). Once the information is printed for every year, the annual average values calculated in *calc_end* (section 7.20.3) are also printed to the 'Results' spreadsheet.

7.20.5 Set and get revenue output

Each of the three revenue-based output parameters has a 'set' subroutine and a 'get' function (e.g. *set_sum_earned_rev* and *get_sum_earned_rev*) in the *revenue_output* object. The 'set' subroutines are sent a value to set to the relevant parameter (i.e. *revenue_loc*). This value is calculated using the 'get' functions, as described in sections 7.7.4 (*revenue.update_rev*) and 7.20.3 (*revenue.calc_end*).

7.21 TECHNICIANS OUTPUT

The output information related to the technicians is controlled and printed by the *technicians_object* and the *techs_output* class module. As described throughout section 7.15, the *technicians_object* not only acts as the control object for technician-based aspects throughout the simulated project lifetime, it also controls the output information. In this regard, it acts as a similar object to *array_output_list* and *wec_output_list* described in sections 7.17 and 7.18 respectively. This includes creating and initialising a *techs_output* object (known as *techs_output_arr*) for each year in the array lifetime, as well as a zero entry to contain averaged information (section 7.15.1). The following procedures in the *technicians_ object* are used to control the output information and are described further in this section:

- *draw*
- *run_title*
- *calc_end*
- *post_process_contractor_fees*

When the *techs_output* objects are initialised by the *start* procedure (section 7.15.1), the contractor fees parameter (*contractor_fees*) is initialised to zero.

7.21.1 Draw

The *draw* function is called at the end of a simulated array lifetime by the *post_process* procedure in the *maint_manager_object* class module (section 7.5.9) in order to control the printing of technicians-based outputs to the 'Results' spreadsheet (section 6.1). The function is sent the reference IDs of the *row* and column (*col*) positions to start printing the information. The headers for the years of the array lifetime are printed by calling the subroutine *run_title* (section 7.21.2). The annual average output values are calculated by calling the subroutine *calc_end* (section 7.21.3). The output information for each year of the project lifetime is printed by calling the *post_process_contractor_fees* procedure (section 7.21.4). The function then returns the ID of the new row to start printing by setting *draw* to be existing value of *row* plus the required gap for formatting purposes.

7.21.2 Run title

The *run_title* subroutine is called by the *draw* function (section 7.21.1) in order to print the year headers, as well as "per year", in the appropriate cells in the *results_sheet*. Each year in the array lifetime (*no_run*) is considered in a *for* loop with the identifier *i*. The header is printed as "year " plus the value of *i*.

7.21.3 Calculate end

The subroutine *calc_end* is called by *draw* (section 7.21.1) in order to fill the zero entry of the *techs_output_arr* list (i.e. *techs_output_arr(0,1)*) with the contractor fees averaged across the lifetime of the project. Each year is considered in turn using a *for* loop with the identifier *i*, from 1 to *no_run.* For each year, the *set_contractor_fees* subroutine in the *techs_output_arr(0,1)* object (section 7.21.6) is called. The argument sent is the current average value (i.e. *get_contractor_fees*) plus the value for the year under consideration (i.e. *techs_output_arr(i, 1).get_contractor_fees*) divided by *no_run*.

7.21.4 Post process contractor fees

The *post_process_contractor_fees* subroutine in the *technicians_object* prints the annual output information for incurred contractor fees. The header "Contractor fees" is printed in the appropriate cell, along with the *output_money_format* (section 7.2.4). Each year of the project lifetime is considered in a *for* loop with the identifier *i*. The output information for that year is then obtained from the relevant *techs_output_arr* object (i.e. *techs_output_arr(i, 1)*, section 7.21.6) and printed to the appropriate cell (i.e. *col + i*). The *output_val_divider* is used to convert the information into the required format defined by the user (section 7.2.4). Once the information is printed for every year, the annual average values calculated in *calc_end* (section 7.21.3) are also printed to the 'Results' spreadsheet.

7.21.5 Add contractor fees

The *add_contractor_fees* subroutine in the *techs_output* class module is called by the procedure of the same name in the *technicians_object* (section 7.15.3) in order to add a particular value (*val_to_add*) to the *contractor_fees*.

7.21.6 Set and get technicians output

The contractor fees output parameter has a 'set' subroutine (*set_contractor_fees*) and a 'get' function (*get_contractor_fees*) in the *techs_output* object. The 'set' subroutine is sent a value (*this_val*) to be the *contractor_fees*. This value is calculated using the 'get' function, as described in section 7.21.3 (*technicians.calc_end*).

7.22 VESSELS OUTPUT

The output information for the vessels are controlled and printed by the *vessel_output_list* class module, using information stored in the *vessel_output*. As described in section 7.8, each vessel listed in the 'Vessels' spreadsheet (section 4.1.3) is assigned its own *vessel_object*. Subsequently, each *vessel_object* has its own *vessel_output_list* class module, which is referred to as *vessel_output_arr*. These output list objects are initialised during the *start* procedures of each *vessel_object* (section 7.8.1). The information for each vessel is printed during the *post_process* procedures in the *maint_manager_object* (section 7.5.9) and *vessel_object* class modules (section 7.8.10). The *calc_total_vessel_costs* subroutine in the *maint_manager_object* also calculates the total output information from all vessels and stores it in the zero entry of the *vessel_object* (i.e. *vessel(0)*), as described in section 7.5.9.

7.22.1 Post process

The *post_process* subroutine of each *vessel_object* is called by *maint_manager* (section 7.5.9) in order to print that vessel's output information to the *results_sheet*. The header is printed by calling the *run_title* subroutine (see section 7.22.5) from the output object (*vessel_output_arr*) once using the condition *if id = 1*. The results are then printed by calling the *draw* subroutine (see section 7.22.4).

7.22.2 Output initialisation

Each *vessel_output_list* class module is created and initialised during the *start* procedure of the associated *vessel_object* (section 7.8.1). The *start* subroutine of *vessel_output_list* is called with the argument *name* (i.e. the name of the vessel). Here, a *vessel_output* object (known as *vessel_output_arr*) is created for each year in the array lifetime, as well as a zero entry to contain averaged information. In each case, the *start* subroutine of the *vessel_output* object is called.

The *start* subroutine of the *vessel_output* class module initialises the output information for that entry by setting the values of *hire_fees, fuel_cost* and *ints_working* (the number of intervals the vessel is used) to zero.

7.22.3 Add fees and intervals working

The vessel output information is updated throughout the simulated lifetime of the wave energy array. This involves calling the subroutine *add_ints_working* in the *vessel_output_list* whenever a

vessel is mobilised (i.e. *vessel.mobilise_boat*, section 7.8.4), as well as the subroutines *add_hire_fees_for_op* and *add_fuel_for_op* whenever a marine operation has taken place (i.e. *vessel. add_op_costs*, section 7.8.8). When these procedures are called, they are sent the current year of the simulation (*irun*) as well as the value to add (e.g. *hire_fees_for_op*). This information is then used to identify the correct output object (i.e. *vessel_output_arr(irun)*) in which to call the relevant 'set' procedure (e.g. *set_hire_fees*, section 7.22.8). The argument sent is the current value of that parameter in the year *irun* (e.g. *get_hire_fees*) plus the value to add (e.g. *hire_fees_for_op*).

7.22.4 Draw

The *draw* subroutine in the *vessel_output_list* is called at the end of a simulated array lifetime by the *post_process* procedure in the *vessel_object* class module (see section 7.22.1) in order to control the printing of vessel-based outputs to the 'Results' spreadsheet (section 6.1). The subroutine is sent the reference IDs of the *row* and column (*col*) positions to start printing the information. The annual average output values are calculated by calling the subroutine *calc_end* (section 7.22.6). The *name* of the vessel is then printed to the relevant cell. The output information for each year of the project lifetime is printed by calling the following procedures in order, updating the *row* to start printing each time (section 7.22.7):

- *post_process_hire_fees*
- *post_process_fuel_cost*
- *post_process_ints_working*

7.22.5 Run title

The *run_title* subroutine (in *vessel_output_list*) is called by the *post_process* procedure in the *vessel_object* class module (see section 7.22.1) in order to print the year headers, as well as "per year", in the appropriate cells in the *results_sheet*. Each year in the array lifetime (*no_run*) is considered in a *for* loop with the identifier *i*. The header is printed as "year " plus the value of *i*. The section header of "Vessels" is also printed in the appropriate cell.

7.22.6 Calculate end

The subroutine *calc_end* (in *vessel_output_list*) is called by *draw* (section 7.22.4) in order to fill the zero entry of the *vessel_output* object (i.e. *vessel_output_arr(0)*) with the values of hire fees, fuel costs and the number of intervals worked, averaged across the lifetime of the project. Each year is considered in turn using a *for* loop with the identifier *i*, from 1 to *no_run.* For each year, the relevant 'set' subroutines (e.g. *set_hire_fees* etc.) in the *vessel_output_arr(0)* object (section 7.22.8) are called. The argument sent in each case is the current average value (e.g. *get_hire_fees*) plus the value for the year under consideration (e.g. *vessel_output_arr(i).get_hire_fees*) divided by *no_run*.

7.22.7 Post process procedures

The three 'post process' subroutines in *vessel_output_list* print the annual output information for incurred hire fees (*post_process_hire_fees*) and fuel costs (*post_process_fuel_cost*), as well as the number of intervals worked (*post_process_ints_working*). The header name (e.g. "Hire fees") is printed in the appropriate cell, along with the *output_money_format* (section 7.2.4) in parentheses. Each year of the project lifetime is considered in a *for* loop with the identifier *i*. The output information for that year is then obtained from the relevant *vessel_output* object (i.e.

vessel_output_arr (i), section 7.22.8) and printed to the appropriate cell (i.e. *col + i*). The *output_val_divider* is used to convert the information into the required format defined by the user (section 7.2.4). Once the information is printed for every year, the annual average values calculated in *calc_end* (section 7.22.6) are also printed to the 'Results' spreadsheet.

7.22.8 Set and get vessel output

Each of the three vessel-based output parameters has a 'set' subroutine and a 'get' function (e.g. *set_hire_fees* and *get_hire_fees*) in the *vessel_output* object. The 'set' subroutines are sent a value to set to the relevant parameter (i.e. *hire_fees_loc*). This value is calculated using the 'get' functions, as described in sections 7.22.3 ('add fees and intervals working') and 7.22.6 (*vessel.calc_end*).

7.23 DELAYS OUTPUT

The output information related to the delays is controlled and printed by the *delays_object* and the *delays_output* class module. As described in section 7.10, the *delays_object* not only acts as the control object for delays-based aspects throughout the simulated project lifetime, it also controls the output information. In this regard, it acts as a similar object to *array_output_list* and *wec_output_list* described in sections 7.17 and 7.18 respectively. This includes creating and initialising a *delays_output* object (known as *delays_output_arr*) for each year in the array lifetime, as well as a zero entry to contain averaged information (section 7.10.1). The following procedures in the *delays_object* are used to control the output information and are described further in this section:

- *draw*
- *run_title*
- *calc_end*
- *post_process_work_attempted*
- *post_process_space_delay*
- *post_process_vessel_delay*
- *post_process_parts_delay*
- *post_process_weather_delay*
- *post_process_techs_delay*

When the *delays_output* objects are initialised by the *start* procedure (section 7.10.1), the six parameters of the number of intervals where work has been attempted (*work_attempted*) and where work is delayed by a lack of O&M base space (*space_delay*), vessels (*vessel_delay*), spare parts (*parts_delay*), suitable weather conditions (*weather_delay*) or technicians (*techs_delay*) are all initialised to zero.

7.23.1 Draw

The *draw* function is called at the end of a simulated array lifetime by the *post_process* procedure in the *maint_manager_object* class module (section 7.5.9) in order to control the printing of delays-based outputs to the 'Results' spreadsheet (section 6.1). The function is sent the reference IDs of the *row* and column (*col*) positions to start printing the information. The headers for the years of the array lifetime are printed by calling the subroutine *run_title* (section 7.23.2). The section header of "Delays" is also printed in the appropriate cell. The annual average output values are calculated by calling the subroutine *calc_end* (section 7.23.3). The output information for each year

of the project lifetime is then printed by calling the following procedures in order, updating the *row* to start printing each time (section 7.23.4):

- *post_process_work_attempted*
- *post_process_space_delay*
- *post_process_vessel_delay*
- *post_process_parts_delay*
- *post_process_weather_delay*
- *post_process_techs_delay*

The printed outputs are formatted into percentage terms by calling the procedure *percent_format* (section 7.23.5). The function returns the ID of the new row to start printing by setting *draw* to be existing value of *row* plus the number of parameters involved in the delays output (i.e. *delays_output_arr(0).get_no_param*).

7.23.2 Run title

The *run_title* subroutine is called by the *draw* function (section 7.23.1) in order to print the year headers, as well as "per year", in the appropriate cells in the *results_sheet*. Each year in the array lifetime (*no_run*) is considered in a *for* loop with the identifier *i*. The header is printed as "year " plus the value of *i*.

7.23.3 Calculate end

The subroutine *calc_end* is called by *draw* (section 7.23.1) in order to fill the zero entry of the *delays_output_arr* list (i.e. *delays_output_arr(0)*) with the number of intervals where work has been attempted and where work is delayed by a lack of O&M base space, vessels, spare parts, suitable weather conditions or technicians, averaged across the lifetime of the project. Each year is considered in turn using a *for* loop with the identifier *i*, from 1 to *no_run.* For each year, the relevant 'set' subroutines (e.g. *set_work_attempted* etc.) in the *delays_output_arr(0)* object (section 7.23.6) are called. The argument sent in each case is the current average value (e.g. *get_work_attempted*) plus the value for the year under consideration (e.g. *delays_output_arr(i).get_work_attempted*) divided by *no_run*.

7.23.4 Post process procedures

The six 'post process' subroutines in the *delays_object* print the annual output information for the number of intervals where work has been attempted (*post_process_work_attempted*) and where work is delayed by a lack of O&M base space (*post_process_space_delay*), vessels (*post_process_vessel_delay*), spare parts (*post_process_parts_delay*), suitable weather conditions (*post_process_weather_delay*) or technicians (*post_process_techs_delay*).

In the subroutine *post_process_work_attempted*, the header "Instances where work was attempted" is printed to the relevant cell. Each year of the project lifetime is considered in a *for* loop with the identifier *i*. The output information (i.e. *get_work_attempted*) for that year is then obtained from the relevant *delays_output_arr* object (i.e. *delays_output_arr(i)*, section 7.23.6) and printed to the appropriate cell (i.e. *col + i*). Once the information is printed for every year, the annual average value calculated in *calc_end* (section 7.23.3) is also printed to the *results_sheet*.

This method is similar in the remaining five 'post process' subroutines. However, the number of intervals where the work is delayed for a specific reason (e.g. lack of space etc.) is converted into a percentage, based on the total number of intervals (*work_attempted*). The header of the subsections therefore say "%" before the cause of the delay (e.g. "% Space delays"). The printed values are first initialised to zero. If any work was delayed in that year (i.e. *delays_output_arr(i).get_work_attempted* is not equal to zero), then the relevant output information (e.g. *get_space_delay*) is divided by the value of *work_attempted*. The resulting value is multiplied by 100 in order to present it in percentage terms.

7.23.5 Percent formatting

The *percent_format* subroutine in *delays_object* is called by the *draw* function (section 7.23.1) in order to present the percentages of delay causes in a readable fashion. To achieve this, the subroutine is sent the range of the printed information (*start_row, end_row, start_col* and *end_col*). It uses this information to set the *NumberFormat* of the appropriate cells to "0.0" (i.e. 1 d.p).

7.23.6 Set and get delays output

Each of the six delays-based output parameters has a 'set' subroutine and a 'get' function (e.g. *set_work_attempted* and *get_work_attempted*) in the *delays_output* object. The 'set' subroutines are sent a value to set to the relevant parameter (e.g. *work_att_loc*). This value is calculated using the 'get' functions, as described in section 7.23.3 (*delays.calc_end*), for example.

7.24 MAINTENANCE MANAGER OUTPUT

Once the control object *maint_manager* has printed the detailed output information for the array to the 'Results' spreadsheet, the *maint_man_output_list* class module is used to control the printing of the summary table described in section 6.1. This actual printing of the summary information is achieved in the *maint_man_output* object, referred to as *maint_man_output_arr* by *maint_man_output_list*.

7.24.1 Start

The *start* subroutine of the *maint_man_output_list* is not called until the detailed outputs have been printed by *post_process* in the *maint_manager_object* (section 7.5.9). It is sent the output objects *array_output_list* (identified as *array_output_arr*, section 7.17), *revenue_object* (*revenue*, which acts as an output list object, section 7.20), *technicians_object* (*techs_object*, acts as an output list, section 7.21) and the *vessel_output_list* corresponding to the total vessel costs (*vessel(0).get_vessel_output*, section 7.22). The purpose of the *start* subroutine is to obtain the annual summary information from the relevant output objects.

The *start* subroutine in *maint_man_output_list* renames the arguments as *array_output*, *revenue*, *techs_object* and *total_vessel_output*. The annual labour cost of the O&M base technicians is then calculated by reading *base_labour_cost* and *overheads_multiplier* directly from the *labour_sheet* (section 4.3). A new *maint_man_output* object (known as *maint_man_output_arr*) is created for each year in the array lifetime, as well as a zero entry to contain averaged information. Each output object is initialised by calling its *start* subroutine with the following arguments:

- *i* - entry of the object in the *maint_man_output_arr* list
- *array_output* - the *array_output_list* object

- *revenue.get_revenue_output(i)* - the *revenue_output* object for the entry *i*
- *base_labour_cost* - annual labour cost of O&M base technicians (including overheads)
- *techs_object.get_techs_output(i)* - the *techs_output* object for the entry *i*
- *total_vessel_output.get_vessel_output(i)* - the *vessel_output* object for all vessels for the entry *i*

The *start* subroutine in each *maint_man_output* then uses these objects to obtain the relevant output information for printing to the summary table. The availability of the wave energy array in that entry (*i_year*) is stored in the variable *avail* after being obtained from the relevant *array_output* object (*array_output_arr.get_array_output(i_year).get_avail*, sections 7.17.9 & 7.19.2). This method of accessing the relevant output object from the output list class module (i.e. *get_array_output*) is not required for the other output information, such as *revenue_output,* due to the format of the arguments sent to the *start* subroutine. In addition to *avail*, the summary output information is stored in the following variables using the relevant objects:

- *sum_rev* - sum of earned revenue (section 7.20)
- *base_labour_cost* - set to be the argument *base_labour_cost_loc*
- *additional_labour_cost* - cost of external contractors (section 7.21)
- *part_cost* - total parts costs (section 7.17)
- *other_cost* - total other costs (section 7.17)
- *inspection_cost* - total inspection costs (section 7.17)
- *vessel_hire_fees* - total vessel hire fees (section 7.22)
- *vessel_fuel_cost* - total vessel fuel costs (section 7.22)

The total operational expenditure incurred in that *i_year* is then calculated by the function *calc_cost*. All the obtained costs are summed in order to calculate this *total_cost*. It should be noted that the obtained values of *part_cost*, *other_cost* and *inspection_cost* were already converted into the user-defined monetary format in the *array_output_list* object (section 7.17). Therefore, these values are multiplied by the *output_val_divider* in order to ensure the function operates in pounds sterling until the information is printed. The *profit* is then calculated by subtracting the *total_cost* from the *sum_rev*.

7.24.2 Draw

The *draw* subroutine of the *maint_man_output_list* is called by the *post_process* procedure in the *maint_manager_object* (section 7.5.9) in order to control the printing of the summary table in the 'Results' spreadsheet (section 6.1). It first calls the *draw_title* subroutine (section 7.24.3) to print the annual headers before calling the *maint_man_output* procedure *print_title* (section 7.24.4) in order to print each of the sub-section headers. A *for* loop then considers each entry in the *maint_man_ output_arr* list and calls *print_data* (section 7.24.5) with the appropriate row and column reference IDs in order to print the summary information in the correct cells.

7.24.3 Draw title

The *draw_title* subroutine (in *maint_man_output_list*) is called by the *draw* procedure (see section 7.24.2) in order to print the year headers, as well as "per year", in the appropriate cells in the *results_sheet*. Each year in the array lifetime (*no_run*) is considered in a *for* loop with the identifier

i. The header is printed as “year “ plus the value of *i*. The table header of “SUMMARY” is also printed in the appropriate cell.

7.24.4 Print titles

The subroutine *print_title* in the *maint_man_output* object is called by the *maint_man_output_list* procedure *draw* (section 7.24.2) in order to print the sub-headers for each output parameter. For every parameter except “Availability”, the *output_money_format* is printed in parentheses alongside the title for clarity.

7.24.5 Print data

The subroutine *print_data* in the *maint_man_output* object is called by the *maint_man_output_list* procedure *draw* (section 7.24.2) in order to print the output data to the summary data. With the exception of *avail*, all the parameters are calculated into pounds sterling format using the *output_val_divider* value. This does not apply to the values of *part_cost*, *other_cost* and *inspection_cost* as they were already converted into the user-defined monetary format in the *array_output_list* object (section 7.17).

7.25 FAILURES OUTPUT

The *failure_output_list* class module is created and initialised by the *run_program* object during the *setup_class* process (section 7.2.4). It is used to control the printing of the output table of fault categories in the ‘Results’ spreadsheet (section 6.1) after the summary table (section 7.24) is complete. The *failure_output_list* is known in *run_program* as *fail_output_list*. The actual printing of the failures output information is achieved in the *failure_output* object, referred to as *fail_output_arr* by *failure_output_list*.

7.25.1 Start

The *start* subroutine in the *failure_output_list* class module is called during the *setup_class* process (section 7.2.4) as shown in Figure 7.2 (page 44). A new *failure_output* object (known as *fail_output_arr*) is created for each fault category defined in the ‘Inputs’ spreadsheet (section 4.1.2). This is achieved by using a *for* loop from zero to the value of *get_no_fail* in the *fail_param_list* class module (section 7.3) with the identifier *i*. The zero entry is included in order to store the total output values of all the fault categories. Each output object (i.e. *fail_output_arr(i)*) is initialised by calling its *start* subroutine with the argument *i*.

The *start* subroutine in each *failure_output* class module initialises each variable to zero. It also sets the *fail_ID* to the argument sent be the calling function. The variables used to store information in the *failure_output* object are:

- *occurrence* - total number of times this failure has occurred
- *occ_repaired* - total number of times this failure has been repaired
- *part_cost* - total parts costs incurred by this failure per year
- *other_cost* - total other costs incurred by this failure per year
- *total_hire_fees* - total vessel hire fees attributed to this failure per year
- *total_fuel_costs* - total vessel fuels costs attributed to this failure per year
- *total_costs* - total OPEX attributed to this failure per year

- *lost_rev_total* - total lost revenue attributed to this failure per year (note; all lost revenue parameters are 'per year')
- *lost_rev_onsite_repair* - total lost revenue attributed to this failure whilst repairs are being undertaken on site
- *lost_rev_in_transit* - total lost revenue attributed to this failure whilst a vessel is in transit
- *lost_rev_offsite* - total lost revenue attributed to this failure whilst the affected WEC is offsite
- *lost_rev_onsite* - total lost revenue attributed to this failure whilst the affected system is on site
- *lost_rev_wait_space* - total lost revenue attributed to this failure whilst retrieval of the affected WEC is delayed due to a lack of space at the O&M base
- *lost_rev_wait_vessel* - total lost revenue attributed to this failure whilst retrieval or installation of the affected WEC is delayed due to a lack of a suitable vessel
- *lost_rev_wait_parts* - total lost revenue attributed to this failure whilst the repair is delayed due to a lack of available spare parts
- *lost_rev_wait_weather* - total lost revenue attributed to this failure whilst retrieval or installation of the affected WEC is delayed due to adverse weather conditions
- *lost_rev_wait_techs* - total lost revenue attributed to this failure whilst repair, retrieval or installation of the affected WEC is delayed due to a lack of available technicians
- *lost_rev_onsite_none* - total lost revenue attributed to this failure whilst the system is on site but not set for repair
- *lost_rev_offsite_none* - total lost revenue attributed to this failure whilst the WEC is undergoing offsite repairs or inspection

7.25.2 Set occurrence and costs

The following four subroutines in the *failure_output_list* object are called throughout the simulated project lifetime, as described primarily in sections 7.13 (*array_object*) and 7.14 (*wec_object*):

- *set_total_occurrence*
- *set_costs_repair*
- *set_vessel_hire_fees*
- *set_vessel_fuel_cost*

The subroutine *set_total_occurrence* is sent the argument *failure_ID* whenever a failure is simulated to have occurred in the *determine_failure* procedures of *array_object* (section 7.13.2) and *wec_object* (section 7.14.2). It uses *failure_ID* to identify the correct *failure_output* object (i.e. *fail_output_arr(failure_ID)*) and calls its procedure *set_total_occurrence* in order to update the value of *occurrence* by adding 1.

The subroutine *set_costs_repair* is called whenever a repair (or series of repairs) has been completed in the *next_interval* procedures of *array_object* (section 7.13.7) and *wec_object* (section 7.14.14). It is sent the list of completed repairs in the variable *fail_arr_id*. Each failure is considered in turn using a *for* loop with the identifier *i*. In each case, the *set_costs_repair* subroutine in the relevant *failure_output* object (i.e. *fail_output_arr(fail_arr_id(i))*) is called. Here, the value of *occ_repaired* is updated by adding 1. In addition, the relevant *fail_param_list* object (i.e. *fail_param_list.get_fail_param(fail_ID)*) is used to obtain the repair costs for that failure (*get_part* and *get_*

other, section 7.3). These values are divided by the number of years in the array lifetime (*no_run*) before being added to the values of *part_cost* and *other_cost* respectively, thereby providing the output information in a 'per year' format.

The subroutines *set_vessel_hire_fees* and *set_vessel_fuel_cost* are both called by the procedure *assign_vessel_costs_output* in the *wec_object* (section 7.14.12) in order to assign a particular cost (*hire_fees_to_add* or *fuel_cost_to_add*) to a certain failure (*this_fail*). In both cases, the relevant *failure_output* object (i.e. *fail_output_arr(this_fail)*) is identified and the appropriate procedure (*set_hire_fees* or *set_fuel_costs*) is called. These subroutines then update the values of *total_hire_fees* and *total_fuel_costs* respectively by adding the cost once it has been divided by *no_run*.

7.25.3 Next_interval

At the end of every interval throughout the simulated array lifetime, the procedure *assign_lost_revenue_fails_maint* in the *array_object* (section 7.13.8) is used to assign lost revenue to failures and maintenance. It achieves this by calling the *next_interval* subroutine in the *failure_output_list* object with the arguments *array_power* (between 0 and 1), *full_rev* (theoretical revenue during this interval if array at 100%) and *store_array* (a 2D array containing information about the failures to assign lost revenue). As described in section 7.13.8, each failure contained in *store_array* has three entries:

1. ID of the failure
2. Portion of lost revenue to be assigned
3. Delay state

The *next_interval* subroutine utilises this information by first calculating the total lost revenue at this interval (i.e. *this_lost_rev = full_rev * (1 - array_power)*). It then loops through each failure listed in *store_array* (i.e. the second dimension) with the identifier *i* and identifies the *state* (i.e. the third entry). The relevant *failure_output* object (i.e. *fail_output_arr(store_array(1, i))*) is identified and its *next_interval* procedure is called. The arguments sent are *this_lost_rev*, *store_array(2, i)* (i.e. the portion to be assigned to that failure) and *state*.

In the *next_interval* subroutine of the *failure_output* object the portion to be assigned to that failure is renamed *this_share*. The 'per year' value of *lost_rev_total* is then updated by adding (*this_lost_rev * this_share) / no_run*. A series of if-else conditions is then used to assign the lost revenue to be appropriate breakdown variable based on the String value *state*. As described in section 7.13.8, the *state* can be given as one of the following entries. The associated variables in *failure_output* are listed alongside the text values.

- "onsite repair" - *lost_rev_onsite_repair*
- "in transit" - *lost_rev_in_transit*
- "offsite" plus;
    - "none" - *lost_rev_offsite_none*
    - "space" - *lost_rev_wait_space*
    - "vessel" - *lost_rev_wait_vessel*
    - "parts" - *lost_rev_wait_parts*
    - "weather" - *lost_rev_wait_weather*
    - "techs" - *lost_rev_wait_techs*
- "onsite" plus;
    - "none" - *lost_rev_onsite_none*

- or one of the delay causes

In the case of *state* being either “offsite” or “onsite” followed by a delay cause, the in-built function *Mid* is used to extract the value of *delay_cause*. If *delay_cause* is “none” then the appropriate variables (i.e. *lost_rev_offsite_none* or *lost_rev_onsite_none*) are updated as before. Otherwise, the subroutine *assign_to_delay* is sent the values of *delay_cause* and *this_lost_rev * this_share* in order to assign the lost revenue to the appropriate delay cause. Error handling is in place if the values of *state* or *delay_cause* are not recognised.

7.25.4 Draw

The *draw* function in the *failure_output_list* object is called by the *post_process* procedure in *run_ program* (section 7.2.5) in order to control the printing of the failure categories output table. It is sent the *row* and column (*col*) reference IDs to start printing in the ‘Results’ spreadsheet (section 6.1). It first calls the *draw_title* subroutine (section 7.25.5) to print the failure name headers (and “total”) before calling the *failure_output* procedure *print_title* (section 7.25.6) in order to print each of the sub-section headers. The subroutine *calc_end* (section 7.25.7) is called in order to sum the values of all failure categories and store them in the zero entry of *fail_output_arr*. A *for* loop then considers each fault category with the identifier *i* and calls *print_data* in the relevant object (*fail_output_arr(i)*, section 7.25.8) with the appropriate row and column reference IDs in order to print the summary information in the correct cells. This subroutine is also called in the zero entry (i.e. *fail_output_arr(0)*) in order to print the total values.

The function name (*draw*) is then updated to be the current value of *row* plus the number of fault categories (i.e. *fail_param_list.get_no_fail*) and a gap for formatting purposes. This enables the *post_process* procedure in *run_ program* (section 7.2.5) to determine which row to start printing the next section.

7.25.5 Draw title

The *draw_title* subroutine (in *failure_output_list*) is called by the *draw* procedure (see section 7.25.4) in order to print the fault category headers, as well as “total”, in the appropriate cells in the *results_sheet*. Each fault category (i.e. *fail_param_list.get_no_fail*, section 7.3) is considered in a *for* loop with the identifier *i*. The header is printed as the name of that category using the function *get_name* in the relevant *fail_param_list* object (i.e. *fail_param_list.get_fail_param(i)*). The header of “total” is also printed in the appropriate cell.

7.25.6 Print titles

The subroutine *print_title* in the *failure_output* object is called by the *failure_output_list* procedure *draw* (section 7.25.4) in order to print the sub-headers of each output parameter. For every parameter except "failure id" and the occurrences, the *output_money_format* is printed in parentheses alongside the title and “per year” for clarity. In the cases of "total occurrence in farm” and "total occurrences repaired”, the number of years in the project lifetime (*no_run*) is also printed to avoid confusion.

A variable *col_to_sort_by* is defined this this subroutine. It is used to determine which column to sort the failure tables by, if required. See section 7.25.10 for further information on how the value of *col_to_sort_by* is used.

7.25.7 Calculate end

The subroutine *calc_end* (in *failure_output_list*) is called by the *draw* procedure (see section 7.25.4) in order to sum the values of each parameter for all fault categories. It also calculates the total operational expenditure attributed to each fault category. To achieve this, a *for* loop considers each fault category in turn with the identifier *i* (up to *fail_param_list.get_no_fail*, section 7.3). In each case, the *calc_total_cost* subroutine in the relevant *failure_output* object (i.e. *fail_output_arr(i)*) is called. The *calc_end* procedure in the zero entry of *fail_output_arr* (i.e. *fail_output_arr(0)*) is then called and sent the argument of the current output object in the loop (i.e. *fail_output_arr(i)*).

The *calc_total_cost* subroutine in the *failure_output* object calculates the total operational expenditure attributed to that failure (*total_costs*) by summing the values of *part_cost, other_cost, total_hire_fees* and *total_fuel_costs*.

The *calc_end* subroutine in the *failure_output* object updates each parameter by adding the value of the same parameter obtained from the output object sent as the argument. But doing this for each entry in *fail_output_arr*, the *failure_output_list* fills the zero entry with the summed values.

7.25.8 Print data

The subroutine *print_data* in the *failure_output* object is called by the *failure_output_list* procedure *draw* (section 7.25.4) in order to print the output data to the summary data. With the exception of *fail_ID, occurrence* and *occ_repaired*, all the parameters are calculated into pounds sterling format using the *output_val_divider* value.

7.25.9 Set and get procedures

Each of the output parameters in the *failure_output* object has a 'get' function (e.g. *get_occurrence, get_occ_repaired* etc.). These functions allow procedures in other objects to access the values of the parameters. An example of this is described in section 7.25.7, where *calc_end* in the zero entry of the list (i.e. *fail_output_arr(0)*) sums the output values of all fault categories.

7.25.10 Sort failure table

At the end of a single simulated lifetime, the table of fault categories in the *result_sheet* is rearranged so that high impact fault categories are immediately identifiable, as described in section 7.2.2. The function *sort_fails_table* in the *failure_output_list* object is used to achieve this. The ID of the column in the failure tables to sort by (*col_to_sort_by*) is obtained from the *failure_output* object (i.e. *fail_output_arr(1).get_col_to_sort_by*). The entire output table of fault categories is selected using the *Range* function with the number of categories identifying the last row (i.e. *fail_param_list.get_no_fail*, section 7.3) and the number of output parameters identifying the last column (i.e. *fail_output_arr(1).get_num_param*). The in-built VBA function *Sort* is then used to rearrange the table in descending order with the values in *col_to_sort_by*.

7.26 MAINTENANCE OUTPUT

The *maint_output_list* class module is created and initialised by the *run_program* object during the *setup_class* process (section 7.2.4). It is used to control the printing of the output table of scheduled maintenance tasks in the 'Results' spreadsheet (section 6.1) after the fault categories output table (section 7.25) is complete. The actual printing of the maintenance output information is achieved in the *maint_output* object, referred to as *maint_output_arr* by *maint_output_list*.

7.26.1 Start

The *start* subroutine in the *maint_output_list* class module is called during the *setup_class* process (section 7.2.4) as shown in Figure 7.2 (page 44). A new *maint_output* object (known as *maint_output_arr*) is created for each scheduled maintenance task defined in the 'Inputs' spreadsheet (section 4.1.3). This is achieved by using a *for* loop from zero to the value of *get_no_maint* in the *maint_param_list* class module (section 7.4) with the identifier *i*. The zero entry is included in order to store the total output values of all the maintenance tasks. Each output object (i.e. *maint_output_arr(i)*) is initialised by calling its *start* subroutine with the argument *i*.

The *start* subroutine in each *maint_output* class module initialises each variable to zero. It also sets the *maint_id* to the argument sent be the calling function. The variables used to store information in the *maint_output* object are:

- *occurrence* - total number of times this maintenance task has been completed
- *part_cost* - total parts costs incurred by this task per year
- *other_cost* - total other costs incurred by this task per year
- *inspection_cost* - total inspection costs incurred by this task per year
- *total_hire_fees* - total vessel hire fees attributed to this task per year
- *total_fuel_costs* - total vessel fuels costs attributed to this task per year
- *total_costs* - total OPEX attributed to this task per year
- *lost_rev_total* - total lost revenue attributed to this task per year (note; all lost revenue parameters are 'per year')
- *lost_rev_onsite_repair* - total lost revenue attributed to this task whilst work is being undertaken on site
- *lost_rev_in_transit* - total lost revenue attributed to this task whilst a vessel is in transit
- *lost_rev_offsite* - total lost revenue attributed to this task whilst the affected WEC is offsite
- *lost_rev_onsite* - total lost revenue attributed to this task whilst the affected system is on site
- *lost_rev_wait_space* - total lost revenue attributed to this task whilst retrieval of the affected WEC is delayed due to a lack of space at the O&M base
- *lost_rev_wait_vessel* - total lost revenue attributed to this task whilst retrieval or installation of the affected WEC is delayed due to a lack of a suitable vessel
- *lost_rev_wait_parts* - total lost revenue attributed to this task whilst the event is delayed due to a lack of available spare parts
- *lost_rev_wait_weather* - total lost revenue attributed to this task whilst retrieval or installation of the affected WEC is delayed due to adverse weather conditions
- *lost_rev_wait_techs* - total lost revenue attributed to this task whilst the work, retrieval or installation of the affected WEC is delayed due to a lack of available technicians

- *lost_rev_onsite_none* - total lost revenue attributed to this task whilst the system is on site but not set to undergo the work
- *lost_rev_offsite_none* - total lost revenue attributed to this task whilst the WEC is undergoing offsite repairs or inspection

7.26.2 Set costs

The following three subroutines in the *maint_output_list* object are called throughout the simulated project lifetime, as described primarily in sections 7.13 (*array_object*) and 7.14 (*wec_object*):

- *set_costs_maint*
- *set_vessel_hire_fees*
- *set_vessel_fuel_cost*

The subroutine *set_costs_maint* is called whenever one or more scheduled maintenance tasks have been completed in the *next_interval* procedures of *array_object* (section 7.13.7) and *wec_object* (section 7.14.14). It is sent the ID of each completed task in the variable *maint_id*. The *set_costs_maint* subroutine in the relevant *maint_output* object (i.e. *maint_output_arr(maint_id)*) is called. Here, the value of *occurrence* is updated by adding 1. In addition, the relevant *maint_param_list* object (i.e. *maint_param_list.get_maint_param(maint_id)*) is used to obtain the costs incurred for that task (*get_part*, *get_other* and *get_inspection*, section 7.4). These values are divided by the number of years in the array lifetime (*no_run*) before being added to the values of *part_cost*, *other_cost* and *inspection_cost* respectively, thereby providing the output information in a 'per year' format.

The subroutines *set_vessel_hire_fees* and *set_vessel_fuel_cost* are both called by the procedure *assign_vessel_costs_output* in the *wec_object* (section 7.14.12) in order to assign a particular cost (*hire_fees_to_add* or *fuel_cost_to_add*) to a certain maintenance task (*this_maint*). In both cases, the relevant *maint_output* object (i.e. *maint_output_arr(this_maint)*) is identified and the appropriate procedure (*set_hire_fees* or *set_fuel_costs*) is called. These subroutines then update the values of *total_hire_fees* and *total_fuel_costs* respectively by adding the cost once it has been divided by *no_run*.

7.26.3 Next_interval

At the end of every interval throughout the simulated array lifetime, the procedure *assign_lost_revenue_fails_maint* in the *array_object* (section 7.13.8) is used to assign lost revenue to failures and maintenance. It achieves this by calling the *next_interval* subroutine in the *maint_output_list* object with the arguments *array_power* (between 0 and 1), *full_rev* (theoretical revenue during this interval if array at 100%) and *store_array* (a 2D array containing information about the maintenance tasks to assign lost revenue). As described in section 7.13.8, each maintenance task contained in *store_array* has three entries:

1. ID of the task
2. Portion of lost revenue to be assigned
3. Delay state

The *next_interval* subroutine utilises this information by first calculating the total lost revenue at this interval (i.e. *this_lost_rev = full_rev * (1 - array_power)*). It then loops through each maintenance task listed in *store_array* (i.e. the second dimension) with the identifier *i* and

identifies the *state* (i.e. the third entry). The relevant *maint_output* object (i.e. *maint_output_arr(store_array(1, i))*) is identified and its *next_interval* procedure is called. The arguments sent are *this_lost_rev*, *store_array(2, i)* (i.e. the portion to be assigned to that task) and *state*.

In the *next_interval* subroutine of the *maint_output* object, the portion to be assigned to that maintenance task is renamed *this_share*. The 'per year' value of *lost_rev_total* is then updated by adding (*this_lost_rev * this_share) / no_run*. A series of if-else conditions is then used to assign the lost revenue to be appropriate breakdown variable based on the String value *state*. As described in section 7.13.8, the *state* can be given as one of the following entries. The associated variables in *maint_output* are listed alongside the text values.

- "onsite repair" - *lost_rev_onsite_repair*
- "in transit" - *lost_rev_in_transit*
- "offsite" plus;
  - "none" - *lost_rev_offsite_none*
  - "space" - *lost_rev_wait_space*
  - "vessel" - *lost_rev_wait_vessel*
  - "parts" - *lost_rev_wait_parts*
  - "weather" - *lost_rev_wait_weather*
  - "techs" - *lost_rev_wait_techs*
- "onsite" plus;
  - "none" - *lost_rev_onsite_none*
  - or one of the delay causes

In the case of *state* being either "offsite" or "onsite" followed by a delay cause, the in-built function *Mid* is used to extract the value of *delay_cause*. If *delay_cause* is "none" then the appropriate variables (i.e. *lost_rev_offsite_none* or *lost_rev_onsite_none*) are updated as before. Otherwise, the subroutine *assign_to_delay* is sent the values of *delay_cause* and *this_lost_rev * this_share* in order to assign the lost revenue to the appropriate delay cause. Error handling is in place if the values of *state* or *delay_cause* are not recognised.

7.26.4 Draw

The *draw* subroutine in the *maint_output_list* object is called by the *post_process* procedure in *run_ program* (section 7.2.5) in order to control the printing of the maintenance tasks output table. It is sent the *row* and column (*col*) reference IDs to start printing in the 'Results' spreadsheet (section 6.1). It first calls the *draw_title* subroutine (section 7.26.5) to print the task name headers (and "total") before calling the *maint_output* procedure *print_title* (section 7.26.6) in order to print each of the sub-section headers. The subroutine *calc_end* (section 7.26.7) is called in order to sum the values of all maintenance tasks and store them in the zero entry of *maint_output_arr*. A *for* loop then considers each maintenance task with the identifier *i* and calls *print_data* in the relevant object (*maint_output_arr(i)*, section 7.26.8) with the appropriate row and column reference IDs in order to print the summary information in the correct cells. This subroutine is also called in the zero entry (i.e. *maint_output_arr(0)*) in order to print the total values.

7.26.5 Draw title

The *draw_title* subroutine (in *maint_output_list*) is called by the *draw* procedure (see section 7.26.4) in order to print the maintenance task headers, as well as "total", in the appropriate cells in

the *results_sheet*. Each maintenance task (i.e. *maint_param_list.get_no_maint*, section 7.4) is considered in a *for* loop with the identifier *i*. The header is printed as the name of that category using the function *get_name* in the relevant *maint_param_list* object (i.e. *maint_param_list.get_maint_param(i)*). The header of “total” is also printed in the appropriate cell.

7.26.6 Print titles

The subroutine *print_title* in the *maint_output* object is called by the *maint_output_list* procedure *draw* (section 7.26.4) in order to print the sub-headers of each output parameter. For every parameter except "maint id" and “occurrence”, the *output_money_format* is printed in parentheses alongside the title and “per year” for clarity. In the case of "occurrence”, the number of years in the project lifetime (*no_run*) is also printed to avoid confusion.

7.26.7 Calculate end

The subroutine *calc_end* (in *maint_output_list*) is called by the *draw* procedure (see section 7.26.4) in order to sum the values of each parameter for all scheduled maintenance tasks. It also calculates the total operational expenditure attributed to each task. To achieve this, a *for* loop considers each maintenance task in turn with the identifier *i* (up to *maint_param_list.get_no_maint*, section 7.4). In each case, the *calc_total_cost* subroutine in the relevant *maint_output* object (i.e. *maint_output_arr(i)*) is called. The *calc_end* procedure in the zero entry of *maint_output_arr* (i.e. *maint_output_arr(0)*) is then called and sent the argument of the current output object in the loop (i.e. *maint_output_arr(i)*).

The *calc_total_cost* subroutine in the *maint_output* object calculates the total operational expenditure attributed to that maintenance task (*total_costs*) by summing the values of *part_cost, other_cost, inspection_cost, total_hire_fees* and *total_fuel_costs*.

The *calc_end* subroutine in the *maint_output* object updates each parameter by adding the value of the same parameter obtained from the output object sent as the argument. But doing this for each entry in *maint_output_arr*, the *maint_output_list* fills the zero entry with the summed values.

7.26.8 Print data

The subroutine *print_data* in the *maint_output* object is called by the *maint_output_list* procedure *draw* (section 7.26.4) in order to print the output data to the summary data. With the exception of *maint_id* and *occurrence*, all the parameters are calculated into pounds sterling format using the *output_val_divider* value.

7.26.9 Set and get procedures

Each of the output parameters in the *maint_output* object has a ‘get’ function (e.g. *get_occurrence, get_part_cost* etc.). These functions allow procedures in other objects to access the values of the parameters. An example of this is described in section 7.26.7, where *calc_end* in the zero entry of the list (i.e. *maint_output_arr(0)*) sums the output values of all scheduled maintenance tasks.

7.27 GRAPH CREATOR

The *graph_creator* class module is used to control the creation of the output graphs described in sections 6.1 and 6.3. It is created by the *run_program* object (section 7.2.1) and called at the end of either a single simulated lifetime (*run_multi*, section 7.2.2) or a statistical run (*stat_run_sub*, section 7.2.7). A number of variables are defined for use throughout the class module:

- *NewChart* - name of a new chart
- *num_runs* - number of simulations completed in a statistical run
- *MyHeight* - required height of graphs
- *MyWidth* - required width of graphs
- *SecHeight* - required height of smaller graphs
- *SecWidth* - required width of smaller graphs
- *chart_text_font* - font of chart text
- *chart_text_size* - size of chart text

The dimensions of the graphs (i.e. *MyHeight, MyWidth, SecHeight* and *SecWidth*) are given to VBA in 'point' format. This can be converted to common units of measurements with the knowledge that there are 72 points per inch and 2.54cm in an inch.

The four key output parameters are defined in a new data type named *my_parameter*:

- *availability*
- *revenue*
- *OPEX*
- *profit*

7.27.1 Master

The *master* subroutine controls the printing of the output graphs and is called at the end of the *run_program* procedures *run_multi* (section 7.2.2) and *stat_run_sub* (section 7.2.7). A number of variables are defined for use throughout the subroutine:

- *wrksheet* - a variable with the data type *Worksheet*
- *avg_start_col* - identifier of the column containing averaged output information
- *lifetime* - number of years in the wave energy array lifetime
- *i* - identifier
- *this_param* - variable with the data type *my_parameter*
- *left_pos* - position of the graph relevant to the left side of a worksheet
- *top_pos* - position of the graph relevant to the top of a worksheet
- *stat_res_sht* - a constant variable identifying the "stat_results" worksheet

If the statistical run has taken place then the "stat_results" worksheet will exist and the output graphs described in section 7.2.7 need to be printed. The smaller dimensions previously defined (*SecHeight* and *SecWidth*) are used to size these graphs. This scenario is identified by looping through each worksheet (*wrksheet*) in the workbook (i.e. the collection of *Worksheets*) and identifying a sheet with the same name as *stat_res_sht*. If the sheet is found then it is activated and all the existing graphs are deleted by calling the custom function *delete_charts* (section 7.1.15). The format of the "stat_results" worksheet (shown in Figure 6.1, page 33) is used to identify the number of years in the simulated lifetime (*lifetime*). This is achieved by using the custom function *find_index_ref* (section 7.1.16) to find the column containing the header "Average" (*avg_start_col*).

The number of simulations completed in the statistical run (*num_runs*) is identified using the in-built function *max* on the first column of the worksheet (i.e. where the ID of the simulations is stored).

As described in section 7.2.7, a graph is created for each parameter in the data type *my_ parameter*. This is achieved by calling the *create_param_graph* subroutine (section 7.27.3) for each entry of a *for* loop covering a certain *this_param*. A *Select Case* condition is used to assign each parameter (*this_param*) an Integer value between 1 and 4 (*i*) in order to calculate the position of the graph in the worksheet (*top_pos*). All four graphs are printed down the left side of the worksheet so the value of *left_pos* does not change. The subroutine *create_param_graph* is sent the arguments *this_ param*, *left_pos*, *top_pos*, and *lifetime*.

A graph showing the cumulative profit over the lifetime of the array is then created by calling the *create_cumul_profit_graph* subroutine (section 7.27.4). The value of *this_param* is set to *profit* and the graph is located at the top of the second column of graphs (i.e. *left_pos = 10 + SecWidth* and *top_pos = 10*). These variables are sent to the subroutine along with *lifetime*.

A histogram showing the monetary values of annual revenue, OPEX and profit is also created by calling the subroutine *create_monetary_histogram* (section 7.27.5). This is to be placed below the cumulative profit graph so the values of *left_pos* and *top_pos* are set to be *10 + SecWidth* and *10 + SecHeight* respectively. The first cell in the *stat_res_sht* worksheet is selected for completion.

Once the graphs in the *stat_res_sht* worksheet have been created (or were not required at all), the subroutine *make_summary_graphs* (section 7.27.6) is called in order to print the charts containing failure and lost revenue information (described in section 6.1).

7.27.2 Insert chart

The function *insert_chart* is used to insert an empty chart of a defined type into the active worksheet. The variable *ChartShape* is defined as the in-built data type *shape*. This enables it to be set to the new chart with the *Shapes* function *AddChart*. The variable name *NewChart* is then set to be the newly added chart so it can be referred to by other procedures in the *graph_creator* object. The argument sent to *insert_chart* (*my_type*) defines which type of the chart to create. If *my_type* is “scatter” then a scatter chart is created by setting the *ChartType* of *NewChart* to be *xlXYScatter-SmoothNoMarkers*. Alternatively, if *my_type* is “histogram” then a histogram is created by setting the *ChartType* of *NewChart* to be *xlColumnClustered*. Error handling is in place if neither of these options is identified. Finally, each existing series in the graph is deleted by looping through each entry in *SeriesCollection* and using the in-built function *Delete*.

7.27.3 Create parameter graphs

The subroutine *create_param_graph* is called by the *master* procedure (section 7.27.1) in order to create a graph for one parameter at a time. The graph shows the selected parameter in terms of the annual average for each year of the array lifetime. This is shown for every simulation undertaken in the statistical run (section 7.2.7) as well as the averaged values. A number of variables are used in the subroutine:

- *this_run* - identifier of the current simulation being considered
- *this_year* - identifier of the current year being considered
- *read_row* - identifier of the row ID to read information from

- *read_col* - identifier of the column ID to read information from
- *series_ID* - ID of the series being added to the graph
- *average_val* - average value obtained from the worksheet
- *x_axis_arr()* - list of the x-axis values
- *y_axis_arr()* - list of the y-axis values

The lists to contain the axis information (*x_axis_arr* and *y_axis_arr*) are both re-sized from 1 to the number of years in the array *lifetime*. The *insert_chart* subroutine (section 7.27.2) is then called with the argument "scatter" so that an empty scatter chart is inserted into the active worksheet. A *for* loop considers each year in the project *lifetime* using the identifier *this_year*. This enables the list of x-axis values to be filled with the year ID (i.e. *x_axis_arr(this_year) = this_year*).

Each simulation undertaken by the statistical run is then considered in a *for* loop with the identifier *this_run* (from 1 to *num_runs*). An extra value is considered (i.e. *num_rows + 1*) in order to include the average values. The row in which the information is stored for that simulation (*read_row*) is identified by taking the format of the worksheet into account (i.e. *this_run + 2*). The *y_axis_arr* list is then filled with the values of the defined parameter (*this_param*) for the simulation under consideration (*this_run*). If the annual values for that run are being considered (i.e. *if this_run <> num_runs + 1*) then the information is read directly from the worksheet. This is achieved by looping through each year (*this_year* from 1 to *lifetime*) and identifying the column containing the information (*read_col*). The format of the worksheet is considered in calculating *read_col*, as well as the numerical value assigned to *this_param* (due to the fact that *my_ parameter* is a custom data type defined using *Enum*). The entry in the list of y-axis values (i.e. *y_axis_arr(this_year)*) is set to be the value of the relevant cell (i.e. *Cells(read_row, read_ col).value*). The exception is for *profit*, where the entry is filled with the revenue value (i.e. in *read_col - 2*) minus the OPEX (i.e. in *read_col - 1*). However, if the value of *this_run* is equal to *num_runs* plus 1, then the average results from all simulations need to be calculated for each year in the *lifetime*. Again, each *this_year* is considered in a *for* loop and the value of *read_col* is identified. If *my_param* is *profit* then the variable *average_val* is first set to be the average revenue (using the in-built function *Average* on the correct *Range*). The average OPEX is then subtracted to store the average profit in *average_val*. If *my_param* is not *profit* then the value of *average_val* is set to be the *Average* of the relevant *Range*. The appropriate entry in the *y_axis_arr* list (i.e. *y_axis_arr(this_year)*) is then set to be *average_val*.

Once the correct values have been calculated, a new series is added to the graph by calling the subroutine *add_this_series* (section 7.27.7) with the relevant arguments (*x_axis_arr* and *y_axis_ arr*). The *series_ID* is identified by counting the number of series' which exist in the graph (i.e. *NewChart.SeriesCollection.count*). The name of this series can then be set as "Run " plus the value of *this_run*. The series then undergoes some formatting. Firstly, the colour of the line is selected using the in-built *ObjectThemeColor* function, distinguishing between *profit* and the other parameters. The lines defining individual simulations (i.e. if *this_run* is not equal to *num_runs + 1*) are then made to be faint and thin. This allows the averaged values (i.e. if *this_run* is equal to *num_runs + 1*) to be the dominant aspect of each graph. In this case, the line is made bolder and thicker. The format of the line is modified to be dashed for *revenue* (i.e. *DashStyle = msoLineSys-Dash*) and dotted for *OPEX* (i.e. *DashStyle = msoLineSysDot*) so that they can be distinguished from the *availability* chart.

After each series has been added, the *format_scatter_chart* subroutine (section 7.27.8) is called in order to position the chart correctly and add the appropriate text information. The final argument is sent as False, indicating to the procedure that this is not the cumulative profit chart.

7.27.4 Create cumulative profit graph

The subroutine *create_cumul_profit_graph* is called by the *master* procedure (section 7.27.1) in order to create a graph showing cumulative profit of the wave energy array throughout its lifetime. Each line on the graph represents a single simulation undertaken in the statistical run (section 7.2.7), as well as the averaged values shown more prominently. The subroutine uses the same variable names as also defined in the *create_param_graph* procedure and it follows a very similar structure (see section 7.27.3). Error handling is in place to ensure that this subroutine is only called when the value of *this_param* is *profit*. The key difference of *create_cumul_profit_graph* compared to *create_param_graph* is that the list of y-axis values (*y_axis_arr*) is filled with the cumulative profit (*cumulative_val*) calculated for each year (*this_year*). The *format_scatter_chart* subroutine (section 7.27.8) is again used in order to position the chart correctly and add the appropriate text information.

7.27.5 Create monetary histogram

The function *create_monetary_histogram* is called by the *master* procedure (section 7.27.1) in order to create a histogram showing the availability, revenue and OPEX averaged over the lifetime of the year and averaged across all simulations undertaken in the statistical run (section 7.2.7). A number of variables are used in the function:

- *i* - identifier
- *average_avail_col* - ID of the column containing the average availability
- *mean_row* - ID of the row containing the mean values from all simulations
- *conf_int_row* - ID of the row containing the 95% confidence intervals
- *x_axis_arr()* - list of the x-axis values
- *y_axis_arr()* - list of the y-axis values
- *x_axis_title* - title of the x-axis
- *y_axis_title* - title of the y-axis
- *theme_colour* - theme colour of the histogram
- *error_rng* - range of 95% confidence intervals
- *series_ID* - ID of the series being added to the histogram

Firstly, the positions of *average_avail_col*, *mean_row* and *conf_int_row* are identified by calling the custom function *find_index_ref* (section 7.1.16) with the appropriate arguments. The list of values (*x_axis_arr* and *y_axis_arr*) are then re-sized to contain three values (i.e. one for each parameter). A *for* loop is then used for each list in order to store the parameter name (in *x_axis_arr*) and the mean values (in *y_axis_arr*). The subroutines *insert_chart* (section 7.27.2) and *add_this_series* (section 7.27.7) are called in order to create a new histogram with the required values (i.e. only one series, *series_ID*). The axis titles (*x_axis_title, y_axis_title*) and *theme_colour* are defined and sent to the *format_histogram* (section 7.27.9) procedure in order to format and position the histogram, as well as adding the appropriate text. One of the arguments sent to the procedure is “smaller”, indicating that the histogram will be the smaller of the pre-defined sizes, therefore matching the scatter diagrams. The 95% confidence intervals are then stored in the variable *error_rng* after being read from the correct row (*conf_int_row*) in the worksheet. The in-built VBA functions

HasErrorBars and *ErrorBars* are used to place the confidence intervals on the histogram and format them. Finally, a small note is added to the histogram (as a *ChartTitle*) to explain that 95% confidence intervals have been applied.

7.27.6 Create summary graphs

The function *make_summary_graphs* is called by the *master* procedure (section 7.27.1) in order to create graphs providing a visual representation of the fault category outputs and causes of lost revenue. If the *graph_creator* class module has been set up at the end of a single simulation then these graphs are printed to the "Results" spreadsheet (section 6.1). If a statistical run process has taken place then the graphs are printed to the "stat_mean" sheet instead (section 6.3). A number of variables are used in the function:

- *wrksheet* - a variable with the data type *Worksheet*
- *sht_name* - name of the worksheet to print the graphs on
- *i* - identifier
- *fail_head_row* - ID of the row containing the headers of the fault categories
- *opex_col* - identifier of the column containing OPEX
- *lost_rev_col* - identifier of the column containing total lost revenue
- *this_col* - identifier
- *x_axis_arr()* - list of the x-axis values
- *y_axis_arr()* - list of the y-axis values
- *left_pos* - position of the graph relevant to the left side of a worksheet
- *top_pos* - position of the graph relevant to the top of a worksheet
- *x_axis_title* - title of the x-axis
- *y_axis_title* - title of the y-axis
- *theme_colour* - theme colour of the histogram
- *this_cause* - identifier
- *count* - counter
- *stat_sht* - a constant variable identifying the "stat_mean" worksheet

Firstly, the appropriate worksheet (*sht_name*) is initialised to be "Results". It stays as this unless the *stat_sht* exists, in which case it becomes "stat_mean". The worksheet is then activated and all existing charts are deleted using the custom function *delete_charts* (section 7.1.15). The positions of *fail_head_row*, *opex_col* and *lost_rev_col* are identified by calling the custom function *find_index_ref* (section 7.1.16) with the appropriate arguments.

The two graphs showing the OPEX and lost revenue attributed to each fault category are then created by setting *this_col* to be *opex_col* first and then *lost_rev_col*. In each case, the lists of the axis values (*x_axis_arr* and *y_axis_arr*) are re-sized to contain information for each of the fault categories (i.e. 1 to *fail_param_list.get_no_fail*, section 7.3). This x-axis values are set to be the fault category ID (in column B) and the y-axis values are read from the relevant row (i.e. *fail_head_row + i*) in *this_col*. The subroutines *insert_chart* (section 7.27.2) and *add_this_series* (section 7.27.7) are called in order to create a new histogram with the required values. The graphs are placed in a column on the left of the worksheet, so *left_pos* is set to be 10 and *top_pos* is calculated based on the value of *this_col* relative to *opex_col*, taking the height of the graph (*MyHeight*) into account. The *x_axis_title* is set to be "Failure ID" for both graphs, whilst the values of *y_axis_title* and *theme_colour* depend on the graph being created. The two fault category charts

are then completed by calling the *format_histogram* procedure (section 7.27.9) in order to format and position the histogram, as well as adding the appropriate text. One of the arguments sent to the procedure is "normal", indicating that the histogram will be sized according to the *MyHeight* and *MyWidth* dimensions previously defined.

The graph to show the different causes of lost revenue is then created. This is achieved by first defining the causes of lost revenue explicitly in the *x_axis_arr* list. The *y_axis_arr* list is then re-sized to match and the counter (*count*) is initialised to -1 (because the subsequently loop starts at zero). A *For Each* loop then considers each entry (*this_cause*) in *x_axis_arr* in turn and adds 1 to *count* so that the entry is allocated a numerical value. The custom function *get_cause_col* (section 7.27.10) is used to obtain the ID of the appropriate column for that cause (*this_col*). This enables the matching entry of *y_axis_arr* to be filled with the total lost revenue (per year) incurred due to that cause of delay. The subroutines *insert_chart* (section 7.27.2) and *add_this_series* (section 7.27.7) are called in order to create a new histogram with the required values. The position-based variables (*left_pos* and *top_pos*) are defined accordingly, as well as the axis titles and *theme_colour*, before being sent to *format_histogram* (section 7.27.9) to complete the chart. The first cell in the worksheet is selected for completion.

7.27.7 Add this series

The function *add_this_series* is called whenever a new series is to be added to the chart being created (*NewChart*). The arguments sent to the function are the lists of x-axis values (*x_axis_arr*) and y-axis values (*y_axis_arr*) used to define the series. The ID of the series (*series_ID*) is identified by counting the number of existing series' once the *NewSeries* is added (i.e. *NewChart.SeriesCollection.count*). The *XValues* and *Values* aspects of the *series_ID* are set to be *x_axis_arr* and *y_axis_arr* respectively.

7.27.8 Format scatter diagram

The function *format_scatter_chart* is called in order to format the current scatter diagram being created (*NewChart*). It is sent the arguments *left_pos* (position from the left edge of the worksheet), *top_pos* (position from the top edge of the worksheet), *lifetime* (number of years in the assessed wave energy array), *this_param* (the output parameter being represented by the graph) and a *Boolean* variable *cumulative* (used to determine when the graph showing cumulative profit is being created). The size of the scatter graph is set to be *SecHeight* and *SecWidth* using the *Parent* functions *Height* and *Width* respectively. The functions *Left* and *Top* are then used in conjunction with *left_pos* and *top_pos* respectively in order to position the chart as specified in the calling procedure. The legend usually accompanying Excel graphs is removed by setting *HasLegend* to *False* in an effort to make the charts more readable. The x-axis title is then set to be "Year" with the *MajorUnit* of 1. The y-axis title is specified using the String variable *str*, which is initially set to the text value of *this_param* (using the function *str_param*, section 7.27.11). The value of *str* is then modified according to the Boolean variable *cumulative*. In addition, if *this_param* is not *availability* then the title also reads the *output_money_format* in parentheses. The y-axis title becomes *str* using the in-built *Axes* and *AxisTitle* functions. The font and size of the text in the chart are set to the pre-defined *chart_text_font* and *chart_text_size* respectively. The limits of the x-axis are set to be 1 and the number of years in the project *lifetime*. It is unnecessary to specify limits for the y-axis due to the large variations expected between assessed WECs and strategies. However, the y-axis upper limit in the case of *this_param* being *availability* is set as 1. The border of the chart

is then removed (i.e. *Border.LineStyle = xlNone*) for presentation purposes. Finally, a *ChartTitle* is added as a replacement for a legend, explaining that the faint lines are the results of individual simulations, whilst the bold line is the average.

7.27.9 Format histogram

The function *format_histogram* is called in order to format the current histogram being created (*NewChart*). It is sent the arguments *left_pos* (position from the left edge of the worksheet), *top_pos* (position from the top edge of the worksheet), *x_axis_title*, *y_axis_title*, *theme_colour* and the *size_type* ("normal" or "smaller"). Firstly, the number of series' contained in the histogram is stored in the variable *series_ID*. The *Parent* functions *Height* and *Width* are then set according to the *size_type* (i.e. *MyHeight* and *MyWidth* if "normal", or *SecHeight* and *SecWidth* if "smaller"). The functions *Left* and *Top* are then used in conjunction with *left_pos* and *top_pos* respectively in order to position the chart as specified in the calling procedure. The legend usually accompanying Excel graphs is removed by setting *HasLegend* to *False* in an effort to make the histogram more readable. Each axis title is added and set to be either *x_axis_title* or *y_axis_title* accordingly. The font and size of the text in the histogram are set to the pre-defined *chart_text_font* and *chart_text_size* respectively. The border of the chart is then removed (i.e. *Border.LineStyle = xlNone*) for presentation purposes. Finally, the *ObjectThemeColor* of the *series_ID* in the histogram is set to be the relevant *msoThemeColorAccent* depending on the value of *theme_colour*.

7.27.10 Get cause column

The function *get_cause_col* is used by the procedure *make_summary_graphs* (section 7.27.6) to identify the ID of the column in a given worksheet (*sht_name*) where the header in the fault categories output table matches a given delay cause (*this_cause*). The ID of the row containing the headers is sent to the function as *fail_head_row*. A *Select Case* condition is used to assess the specified text defined by *this_cause* where a meaningful and concise explanation of the delay cause is stated. The value of *this_cause* is used to identify what the corresponding header for that parameter says in the worksheet exactly. The text of the identified header is stored in the variable *search_text*. The custom function *find_index_ref* (section 7.1.16) is then sent this value, along with the other relevant arguments such as *fail_head_row*, in order to identify the correct column (*get_cause_col*).

7.27.11 String parameter

The function *str_param* is used by the *format_scatter_chart* procedure (section 7.27.8) to identify the String value of the *this_param* argument. The argument will have the custom data type *my_parameter* but has a numerical value in VBA. If VBA sets an axis title to *my_param* directly then it will simply read the numerical value. Therefore, *str_param* uses a *Select Case* condition to convert *my_param* into a return value (*temp_str*) with the String data type. The function name (*str_param*) is then set to be *temp_str* so it can be used by the calling procedure.

8 DOCUMENT INDEX

This section aims to provide the user with a point of reference when viewing the model VBA code. There are a lot of modules, class modules and procedures involved in the code which all serve a particular purpose. Some procedures may only be used once whilst others are utilised time and time again. This section provides a list of the VBA objects (modules and class modules) in alphabetical order. Within each subsection, a list of the procedures (subroutines and functions) is given, also in alphabetical order. Alongside each entry there is the number of the most relevant section and the page number it starts at. Please note that the entry may be discussed in other sections of the document and cross referencing has been applied throughout the report where relevant. This index does not highlight where variables are discussed, even those which are used multiple times, as the user can refer to the relevant section where the variables appear.

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
array_fail	-	7.16.2	122
array_fail	get_fail_id	7.16.2	122
array_fail	start	7.16.2	122
array_fail_list	-	7.16.1	121
array_fail_list	add_fail	7.16.1	121
array_fail_list	get_fail_arr_id	7.16.1	121
array_fail_list	get_fail_number	7.16.1	121
array_fail_list	get_total_fails	7.16.1	121
array_fail_list	start	7.16.1	121
array_fail_list	string_fail	7.16.1	121
array_object	-	7.13	76
array_object	add_array_fail	7.13.2	77
array_object	assign_lost_revenue_fails_maint	7.13.8	84
array_object	attempt_fix	7.13.4	79
array_object	calc_fail_share	7.13.11	88
array_object	count_wecs_offsite	7.13.3	78
array_object	determine_failure	7.13.2	77
array_object	determine_fix	7.13.3	78
array_object	fails_power_loss	7.13.9	87
array_object	fails_power_loss_retrieve	7.13.10	87
array_object	get_num_wecs	7.13.14	89
array_object	get_offshore_hours_subsea	7.13.6	82
array_object	get_technicians_object	7.13.14	89
array_object	get_wec	7.13.14	89
array_object	next_interval	7.13.7	83
array_object	post_process	7.13.13	89
array_object	print_interval	7.13.12	88
array_object	start	7.13.1	77
array_object	sum_wecs_power	7.13.5	82
array_object	update_array_power	7.13.5	82
array_output_list	-	7.17	124
array_output_list	add_maint_costs	7.17.2	124
array_output_list	avail_add	7.17.3	124

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
array_output_list	calc_end	7.17.5	125
array_output_list	draw	7.17.4	125
array_output_list	draw_all_wecs	7.17.7	126
array_output_list	fail_costs	7.17.2	124
array_output_list	get_array_output	7.17.9	126
array_output_list	get_no_param	7.17.9	126
array_output_list	get_total_inspection_costs	7.17.8	126
array_output_list	get_total_other_costs	7.17.8	126
array_output_list	get_total_parts_costs	7.17.8	126
array_output_list	post_process_avail	7.17.6	125
array_output_list	post_process_inspection_cost	7.17.6	125
array_output_list	post_process_other_cost	7.17.6	125
array_output_list	post_process_part_cost	7.17.6	125
array_output_list	set_total_inspection_costs	7.17.8	126
array_output_list	set_total_other_costs	7.17.8	126
array_output_list	set_total_parts_costs	7.17.8	126
array_output_list	start	7.17.1	124
cost_benefit_analysis	-	7.12	71
cost_benefit_analysis	create_full_list	7.12.2	72
cost_benefit_analysis	order_this_list	7.12.5	75
cost_benefit_analysis	start	7.12.1	72
cost_benefit_analysis	worth_repairing_WEC	7.12.4	75
cost_benefit_analysis	worth_retrieving_WEC	7.12.3	73
delays_object	-	7.10	66
delays_object	add_this_delay	7.10.2	66
delays_object	calc_end	7.23.3	136
delays_object	draw	7.23.1	135
delays_object	percent_format	7.23.5	137
delays_object	post_process_parts_delay	7.23.4	136
delays_object	post_process_space_delay	7.23.4	136
delays_object	post_process_techs_delay	7.23.4	136
delays_object	post_process_vessel_delay	7.23.4	136
delays_object	post_process_weather_delay	7.23.4	136
delays_object	post_process_work_attempted	7.23.4	136
delays_object	run_title	7.23.2	136
delays_object	start	7.10.1	66
delays_output	-	7.23	135
delays_output	get_no_param	7.23.6	137
delays_output	get_parts_delay	7.23.6	137
delays_output	get_space_delay	7.23.6	137
delays_output	get_techs_delay	7.23.6	137
delays_output	get_vessel_delay	7.23.6	137
delays_output	get_weather_delay	7.23.6	137
delays_output	get_work_attempted	7.23.6	137
delays_output	set_parts_delay	7.23.6	137
delays_output	set_space_delay	7.23.6	137
delays_output	set_techs_delay	7.23.6	137

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
delays_output	set_vessel_delay	7.23.6	137
delays_output	set_weather_delay	7.23.6	137
delays_output	set_work_attempted	7.23.6	137
delays_output	start	7.23	135
failure_no_object	-	7.16.5	123
failure_no_object	count	7.16.5	123
failure_no_object	ret_intermediate_count	7.16.5	123
failure_no_object	ret_major_count	7.16.5	123
failure_no_object	ret_minor_count	7.16.5	123
failure_no_object	start	7.16.5	123
failure_output	-	7.25	139
failure_output	assign_to_delay	7.25.3	141
failure_output	calc_end	7.25.7	143
failure_output	calc_total_cost	7.25.7	143
failure_output	get_col_to_sort_by	7.25.10	143
failure_output	get_lost_rev_in_transit	7.25.9	143
failure_output	get_lost_rev_offsite	7.25.9	143
failure_output	get_lost_rev_offsite_none	7.25.9	143
failure_output	get_lost_rev_onsite	7.25.9	143
failure_output	get_lost_rev_onsite_none	7.25.9	143
failure_output	get_lost_rev_onsite_repair	7.25.9	143
failure_output	get_lost_rev_total	7.25.9	143
failure_output	get_lost_rev_wait_parts	7.25.9	143
failure_output	get_lost_rev_wait_space	7.25.9	143
failure_output	get_lost_rev_wait_techs	7.25.9	143
failure_output	get_lost_rev_wait_vessel	7.25.9	143
failure_output	get_lost_rev_wait_weather	7.25.9	143
failure_output	get_num_param	7.25.9	143
failure_output	get_occ_repaired	7.25.9	143
failure_output	get_occurrence	7.25.9	143
failure_output	get_other_cost	7.25.9	143
failure_output	get_part_cost	7.25.9	143
failure_output	get_total_costs	7.25.9	143
failure_output	get_total_fuel_costs	7.25.9	143
failure_output	get_total_hire_fees	7.25.9	143
failure_output	next_interval	7.25.3	141
failure_output	print_data	7.25.8	143
failure_output	print_title	7.25.6	142
failure_output	set_costs_repair	7.25.2	140
failure_output	set_fuel_costs	7.25.2	140
failure_output	set_hire_fees	7.25.2	140
failure_output	set_total_occurrence	7.25.2	140
failure_output	start	7.25.1	139
failure_output_list	-	7.25	139
failure_output_list	calc_end	7.25.7	143
failure_output_list	draw	7.25.4	142
failure_output_list	draw_title	7.25.5	142

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
failure_output_list	next_interval	7.25.3	141
failure_output_list	set_costs_repair	7.25.2	140
failure_output_list	set_total_occurrence	7.25.2	140
failure_output_list	set_vessel_fuel_cost	7.25.2	140
failure_output_list	set_vessel_hire_fees	7.25.2	140
failure_output_list	sort_fails_table	7.25.10	143
failure_output_list	start	7.25.1	139
failure_param	-	7.3	48
failure_param	error_finder	7.3.2	49
failure_param	get_action_reqd	7.3.1	48
failure_param	get_days_onshore	7.3.1	48
failure_param	get_hours_offshore	7.3.1	48
failure_param	get_name	7.3.1	48
failure_param	get_ops_limits_type	7.3.1	48
failure_param	get_other	7.3.1	48
failure_param	get_part	7.3.1	48
failure_param	get_percent	7.3.1	48
failure_param	get_power	7.3.1	48
failure_param	get_relevance	7.3.1	48
failure_param	get_severity	7.3.1	48
failure_param	get_techs_reqd	7.3.1	48
failure_param	get_vessel_reqd	7.3.1	48
failure_param	start	7.3.1	48
failure_param_list	-	7.3	48
failure_param_list	get_fail_param	7.3.1	48
failure_param_list	get_no_fail	7.3.1	48
failure_param_list	start	7.3.1	48
functions	-	7.1	36
functions	Col_Letter	7.1.14	39
functions	delete_charts	7.1.15	40
functions	delete_run_sh	7.1.4	37
functions	delete_stat_shts	7.1.5	37
functions	delete_this_sht	7.1.6	37
functions	find_index_ref	7.1.16	40
functions	get_ordered_array_2d	7.1.12	38
functions	insert_sheet	7.1.2	36
functions	is_in_array	7.1.9	38
functions	num_rows	7.1.11	38
functions	max	7.1.7	37
functions	min	7.1.7	37
functions	round_all_decimals	7.1.17	40
functions	str_2d_array	7.1.10	38
functions	str_array	7.1.10	38
functions	terminate_program	7.1.8	38
functions	timer	7.1.3	36
functions	WorkbookOpen	7.1.13	39
graph_creator	-	7.27	148

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
graph_creator	add_this_series	7.27.7	153
graph_creator	create_cumul_profit_graph	7.27.4	151
graph_creator	create_monetary_histogram	7.27.5	151
graph_creator	create_param_graph	7.27.3	149
graph_creator	format_histogram	7.27.9	154
graph_creator	format_scatter_chart	7.27.8	153
graph_creator	get_cause_col	7.27.10	154
graph_creator	insert_chart	7.27.2	149
graph_creator	make_summary_graphs	7.27.6	152
graph_creator	master	7.27.1	148
graph_creator	str_param	7.27.11	154
hindcast_object	-	7.11	67
hindcast_object	get_estimated_monthly_rev	7.11.5	71
hindcast_object	get_estimated_months_install_wait	7.11.5	71
hindcast_object	int_wndo_open	7.11.3	70
hindcast_object	is_daylight	7.11.4	70
hindcast_object	rounded_val	7.11.2	69
hindcast_object	start	7.11.1	67
hindcast_object	this_daylight_wndo_open	7.11.4	70
hindcast_object	this_wndo_open	7.11.3	70
maint_man_output	-	7.24	137
maint_man_output	calc_cost	7.24.1	137
maint_man_output	get_no_param	7.24	137
maint_man_output	print_data	7.24.5	139
maint_man_output	print_title	7.24.4	139
maint_man_output	start	7.24.1	137
maint_man_output_list	-	7.24	137
maint_man_output_list	draw	7.24.2	138
maint_man_output_list	draw_title	7.24.3	138
maint_man_output_list	get_no_param	7.24	137
maint_man_output_list	start	7.24.1	137
maint_manager_object	-	7.5	50
maint_manager_object	calc_total_vessel_costs	7.5.9	53
maint_manager_object	determine_actual_fix	7.5.6	51
maint_manager_object	determine_failure	7.5.4	51
maint_manager_object	determine_fix	7.5.5	51
maint_manager_object	insert_sheet_maint_man	7.5.3	51
maint_manager_object	next_interval	7.5.8	52
maint_manager_object	post_process	7.5.9	53
maint_manager_object	print_interval	7.5.7	52
maint_manager_object	start	7.5.2	50
maint_output	-	7.26	144
maint_output	assign_to_delay	7.26.3	145
maint_output	calc_end	7.26.7	147
maint_output	calc_total_cost	7.26.7	147
maint_output	get_inspection_cost	7.26.9	147
maint_output	get_lost_rev_in_transit	7.26.9	147

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
maint_output	get_lost_rev_offsite	7.26.9	147
maint_output	get_lost_rev_offsite_none	7.26.9	147
maint_output	get_lost_rev_onsite	7.26.9	147
maint_output	get_lost_rev_onsite_none	7.26.9	147
maint_output	get_lost_rev_onsite_repair	7.26.9	147
maint_output	get_lost_rev_total	7.26.9	147
maint_output	get_lost_rev_wait_parts	7.26.9	147
maint_output	get_lost_rev_wait_space	7.26.9	147
maint_output	get_lost_rev_wait_techs	7.26.9	147
maint_output	get_lost_rev_wait_vessel	7.26.9	147
maint_output	get_lost_rev_wait_weather	7.26.9	147
maint_output	get_num_param	7.26.9	147
maint_output	get_occurrence	7.26.9	147
maint_output	get_other_cost	7.26.9	147
maint_output	get_part_cost	7.26.9	147
maint_output	get_total_cost	7.26.9	147
maint_output	get_total_fuel_costs	7.26.9	147
maint_output	get_total_hire_fees	7.26.9	147
maint_output	next_interval	7.26.3	145
maint_output	print_data	7.26.8	147
maint_output	print_title	7.26.6	147
maint_output	set_costs_maint	7.26.2	145
maint_output	set_fuel_costs	7.26.2	145
maint_output	set_hire_fees	7.26.2	145
maint_output	start	7.26.1	144
maint_output_list	-	7.26	144
maint_output_list	calc_end	7.26.7	147
maint_output_list	draw	7.26.4	146
maint_output_list	draw_title	7.26.5	146
maint_output_list	next_interval	7.26.3	145
maint_output_list	set_costs_maint	7.26.2	145
maint_output_list	set_vessel_fuel_cost	7.26.2	145
maint_output_list	set_vessel_hire_fees	7.26.2	145
maint_output_list	start	7.26.1	144
maintenance_param	-	7.4	49
maintenance_param	get_action_reqd	7.4.1	49
maintenance_param	get_days_onshore	7.4.1	49
maintenance_param	get_hours_offshore	7.4.1	49
maintenance_param	get_inspection	7.4.1	49
maintenance_param	get_interval_yrs	7.4.1	49
maintenance_param	get_name	7.4.1	49
maintenance_param	get_ops_limits_type	7.4.1	49
maintenance_param	get_other	7.4.1	49
maintenance_param	get_part	7.4.1	49
maintenance_param	get_relevance	7.4.1	49
maintenance_param	get_staggered_maint	7.4.1	49
maintenance_param	get_techs_reqd	7.4.1	49

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
maintenance_param	get_time_of_year	7.4.1	49
maintenance_param	get_vessel_reqd	7.4.1	49
maintenance_param	start	7.4.1	49
maintenance_param_list	-	7.4	49
maintenance_param_list	get_maint_param	7.4.1	49
maintenance_param_list	get_no_maint	7.4.1	49
maintenance_param_list	start	7.4.1	49
parts_object	-	7.9	63
parts_object	all_parts_available	7.9.2	64
parts_object	correct_type_name	7.9.5	65
parts_object	get_num_parts_types	7.9.1	63
parts_object	multi_parts_types_available	7.9.4	65
parts_object	multi_replacement_types_arr	7.9.3	65
parts_object	next_interval	7.9.7	65
parts_object	order_new_parts	7.9.2	64
parts_object	print_interval	7.9.8	66
parts_object	start	7.9.1	63
parts_object	this_type_id	7.9.6	65
revenue_object	-	7.7	57
revenue_object	draw	7.20.1	130
revenue_object	calc_end	7.20.3	131
revenue_object	get_month	7.7.6	59
revenue_object	get_num_entries	7.7.1	57
revenue_object	get_power	7.7.2	58
revenue_object	get_revenue	7.7.3	58
revenue_object	get_revenue_output	7.7.3	58
revenue_object	get_tariff	7.7.3	58
revenue_object	post_process_earned_rev	7.20.4	131
revenue_object	post_process_lost_rev	7.20.4	131
revenue_object	post_process_theory_rev	7.20.4	131
revenue_object	start	7.7.1	57
revenue_object	revenue_estimate	7.7.6	59
revenue_object	run_title	7.20.2	130
revenue_object	update_rev	7.7.4	58
revenue_output	-	7.20	130
revenue_output	get_no_param	7.20	130
revenue_output	get_sum_earned_rev	7.20.5	131
revenue_output	get_sum_lost_rev	7.20.5	131
revenue_output	get_sum_theory_rev	7.20.5	131
revenue_output	set_sum_earned_rev	7.20.5	131
revenue_output	set_sum_lost_rev	7.20.5	131
revenue_output	set_sum_theory_rev	7.20.5	131
revenue_output	start	7.20	130
run_program	-	7.2	41
run_program	copy_weather_data	7.2.6	45
run_program	fast_run_sub	7.2.2	42
run_program	full_run_sub	7.2.2	42

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
run_program	post_process	7.2.5	45
run_program	setup_class	7.2.4	43
run_program	stat_run_sub	7.2.7	46
run_program	run_multi	7.2.2	42
run_program	run_om	7.2.3	42
technicians_object	-	7.15	118
technicians_object	add_contractor_fees	7.15.3	119
technicians_object	add_tech_working	7.15.2	119
technicians_object	calc_end	7.21.3	132
technicians_object	draw	7.21.1	132
technicians_object	get_annual_labour_cost	7.15.6	120
technicians_object	get_num_techs_avail	7.15.6	120
technicians_object	get_tech_availability	7.15.6	120
technicians_object	get_techs_output	7.15.6	120
technicians_object	next_interval	7.15.4	120
technicians_object	post_process_contractor_fees	7.21.4	132
technicians_object	print_interval	7.15.5	120
technicians_object	run_title	7.21.2	132
technicians_object	start	7.15.1	118
technicians_object	update_contractors_on_hire	7.15.3	119
techs_output	-	7.21	131
techs_output	add_contractor_fees	7.21.5	133
techs_output	get_contractor_fees	7.21.6	133
techs_output	set_contractor_fees	7.21.6	133
techs_output	start	7.21	131
vessel_object	-	7.8	60
vessel_object	add_op_costs	7.8.8	62
vessel_object	calc_fuel_for_op	7.8.7	62
vessel_object	calc_hire_fees_for_op	7.8.6	61
vessel_object	check_availability	7.8.3	61
vessel_object	demobilise_boat	7.8.5	61
vessel_object	get_day_hire_fee	7.8.2	61
vessel_object	get_free_travel_time	7.8.2	61
vessel_object	get_fuel_cost_hr	7.8.2	61
vessel_object	get_id	7.8.2	61
vessel_object	get_name	7.8.2	61
vessel_object	get_num_ints_left_in_use	7.8.2	61
vessel_object	get_num_print_cols	7.8.2	61
vessel_object	get_personnel_capacity	7.8.2	61
vessel_object	get_state	7.8.2	61
vessel_object	get_tow_time	7.8.2	61
vessel_object	get_vessel_output	7.8.2	61
vessel_object	mobilise_boat	7.8.4	61
vessel_object	next_interval	7.8.4	61
vessel_object	num_new_days	7.8.6	61
vessel_object	post_process	7.8.10	62
vessel_object	print_interval	7.8.9	62

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
vessel_object	start	7.8.1	60
vessel_output	-	7.22	133
vessel_output	get_fuel_cost	7.22.8	135
vessel_output	get_hire_fees	7.22.8	135
vessel_output	get_ints_working	7.22.8	135
vessel_output	get_no_param	7.22.8	135
vessel_output	set_fuel_cost	7.22.8	135
vessel_output	set_hire_fees	7.22.8	135
vessel_output	set_ints_working	7.22.8	135
vessel_output	start	7.22.2	133
vessel_output_list	-	7.22	133
vessel_output_list	add_fuel_for_op	7.22.3	133
vessel_output_list	add_hire_fees_for_op	7.22.3	133
vessel_output_list	add_ints_working	7.22.3	133
vessel_output_list	calc_end	7.22.6	134
vessel_output_list	draw	7.22.4	134
vessel_output_list	get_no_param	7.22	133
vessel_output_list	get_vessel_output	7.22	133
vessel_output_list	post_process_fuel_cost	7.22.7	134
vessel_output_list	post_process_hire_fees	7.22.7	134
vessel_output_list	post_process_ints_working	7.22.7	134
vessel_output_list	run_title	7.22.5	134
vessel_output_list	start	7.22.2	133
weather_object	-	7.6	54
weather_object	get_num_ops_lims_types	7.6.1	54
weather_object	get_this_wndo	7.6.2	55
weather_object	is_daylight	7.6.3	56
weather_object	longest_daylight_wndo	7.6.5	56
weather_object	print_interval	7.6.4	56
weather_object	start	7.6.1	54
wec_fail	-	7.16.4	123
wec_fail	get_fail_id	7.16.4	123
wec_fail	start	7.16.4	123
wec_fail_list	-	7.16.3	122
wec_fail_list	add_fail	7.16.3	122
wec_fail_list	get_fail_arr_id	7.16.3	122
wec_fail_list	get_fail_number	7.16.3	122
wec_fail_list	get_total_fails	7.16.3	122
wec_fail_list	start	7.16.3	122
wec_fail_list	string_fail	7.16.3	122
wec_fail_list	update_fail_arr	7.16.3	122
wec_object	-	7.14	89
wec_object	add_wec_fail	7.14.2	92
wec_object	any_fails_need_retrieval	7.14.6	97
wec_object	any_maint_delay	7.14.35	117
wec_object	any_maint_due	7.14.34	116
wec_object	any_maint_ready	7.14.34	116

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
wec_object	assign_offsite_fail_techs	7.14.18	107
wec_object	assign_offsite_maint_techs	7.14.18	107
wec_object	assign_vessel_costs_output	7.14.12	100
wec_object	attempt_fix	7.14.4	93
wec_object	calc_intervals_offsite	7.14.9	98
wec_object	calmest_lims_for_op	7.14.8	98
wec_object	define_set_for_maint	7.14.3	92
wec_object	determine_failure	7.14.2	92
wec_object	fail_currently_under_repair	7.14.19	108
wec_object	fails_time_share_arr	7.14.13	101
wec_object	find_fails_array_position	7.14.19	108
wec_object	find_install_vessel_id	7.14.31	114
wec_object	find_maint_array_position	7.14.19	108
wec_object	full_wndo_open	7.14.11	99
wec_object	get_arr_maint_ready	7.14.17	107
wec_object	get_arr_retrieval_fails	7.14.16	107
wec_object	get_delay_status	7.14.36	117
wec_object	get_fail_list	7.14.36	117
wec_object	get_install_time	7.14.30	114
wec_object	get_maint_interval_in_year	7.14.32	115
wec_object	get_maint_due	7.14.36	117
wec_object	get_max_severity	7.14.27	112
wec_object	get_num_wec_maint_cats	7.14.36	117
wec_object	get_num_wec_maints_due	7.14.34	116
wec_object	get_num_wec_maints_ready	7.14.34	116
wec_object	get_state	7.14.36	117
wec_object	get_time_until_repaired	7.14.29	113
wec_object	get_total_other_costs	7.14.26	112
wec_object	get_total_parts_costs	7.14.26	112
wec_object	get_wec_maint_cat	7.14.36	117
wec_object	get_wec_output_list	7.14.36	117
wec_object	get_wec_power	7.14.36	117
wec_object	intermediate_failures	7.14.28	113
wec_object	ints_to_next_maint	7.14.32	115
wec_object	longest_time_offshore	7.14.7	97
wec_object	maint_currently_being_done	7.14.19	108
wec_object	major_failures	7.14.28	113
wec_object	next_interval	7.14.14	102
wec_object	num_onsite_techs_reqd	7.14.5	96
wec_object	num_retrieval_fails	7.14.33	116
wec_object	print_interval	7.14.20	109
wec_object	ret_action_fails	7.14.25	112
wec_object	ret_action_onsite_priority	7.14.22	111
wec_object	ret_actions_reqd	7.14.21	110
wec_object	ret_part_to_replace	7.14.10	99
wec_object	ret_vessel_id_to_use	7.14.24	112
wec_object	set_maint_due	7.14.3	92

OBJECT NAME	PROCEDURE NAME	MOST RELEVANT SECTION	STARTING PAGE NUMBER OF SECTION
wec_object	start	7.14.1	91
wec_object	this_maint_ready	7.14.34	116
wec_object	try_assign_replacement_parts	7.14.15	106
wec_object	vessel_for_action	7.14.23	111
wec_output	-	7.19	129
wec_output	costs_add	7.19.3	129
wec_output	get_avail	7.19.2	129
wec_output	get_inspection_cost	7.19.2	129
wec_output	get_no_param	7.19.4	129
wec_output	get_other_cost	7.19.2	129
wec_output	get_part_cost	7.19.2	129
wec_output	maint_costs_add	7.19.3	129
wec_output	set_avail	7.19.2	129
wec_output	set_inspection_cost	7.19.2	129
wec_output	set_other_cost	7.19.2	129
wec_output	set_part_cost	7.19.2	129
wec_output	start	7.19.1	129
wec_output_list	-	7.18	127
wec_output_list	add_maint_costs	7.18.2	127
wec_output_list	avail_add	7.18.3	127
wec_output_list	calc_end	7.18.5	128
wec_output_list	draw	7.18.4	128
wec_output_list	fail_costs	7.18.2	127
wec_output_list	get_no_param	7.18.8	129
wec_output_list	get_wec_output	7.18.8	129
wec_output_list	post_process_avail	7.18.6	128
wec_output_list	post_process_inspection_cost	7.18.6	128
wec_output_list	post_process_other_cost	7.18.6	128
wec_output_list	post_process_part_cost	7.18.6	128
wec_output_list	run_title	7.18.7	128
wec_output_list	start	7.18.1	127

9 REFERENCES

DET NORSKE VERITAS (DNV). (2012) Failure Mode and Effect Analysis (FMEA) of Redundant Systems. Recommended Practice. Report ID: DNV-RP-D102.

GRAY, A. M. (2017) *Modelling Operations and Maintenance Strategies for Wave Energy Arrays*. Thesis (EngD). College of Engineering, Mathematics and Physical Sciences, University of Exeter.

WAVE ENERGY SCOTLAND (WES). (2017a) *Operations and Maintenance Simulation Tool - Weather Simulation Report.*

WAVE ENERGY SCOTLAND (WES). (2017b) *Operations and Maintenance Simulation Tool - User Guide.*

WAVE ENERGY SCOTLAND (WES). (2017c) *Operations and Maintenance Simulation Tool - Future Upgrades.*

10 APPENDICES

10.1 LIST OF FIGURES

10.2 LIST OF TABLES

10.3 APPENDIX A

Using an FMEA to obtain O&M tool inputs

A Failure Modes and Effect Analysis (FMEA) of a wave energy converter (WEC) will provide a complete list of the components within each subsystem of the device. Every possible failure mode of each component is assessed in terms of its likelihood and consequence. The FMEA process is used substantially in industrial to identify areas of the design requiring further mitigation in order to reduce risk. The likelihood values are usually chosen from probability bands, due to the fact that exact failure rate information is difficult to obtain (although it is preferred). The O&M model utilises failure rate data in the form of probability of failure per year. If the data is obtained in a different format during the FMEA then appropriate conversions need to be undertaken. Useful equations for this process are:

$$\lambda = \frac{T_f}{t} \qquad \text{(A.1)}$$

$$MTBF = \frac{1}{\lambda} \qquad \text{(A.2)}$$

$$F = 1 - e^{-\lambda} \qquad \text{(A.3)}$$

$$R = 1 - F \qquad \text{(A.4)}$$

Where:

- λ = annual failure rate
- T_f = total number of expected failures in design lifetime
- t = design lifetime
- F = annual probability of failure
- R = annual reliability (i.e. probability of not failing)

The defined likelihood of each failure mode leads onto calculating the probability of failure, whilst the consequence indicates the severity of the fault. A wave energy converter may consist of many hundreds of different components. It is unfeasible to apply the Monte Carlo analysis of the O&M model to each component due to the unacceptable computational time required. Instead, the WEC components are grouped into fault categories representing the main engineering aspects of a wave energy machine; hydraulics, moorings, structural and electrical. The fault categories also represent the severity of component failure, in terms of cost and time to repair, and are therefore classed as major, intermediate or minor. Expert judgement must be used throughout this process to ensure that each component is represented by the appropriate fault category. When calculating the probability of failure of a fault category, the following equation must be used:

$$F = 1 - \prod_{i-1}^{n} Ri^{N_i} \qquad \text{(A.5)}$$

Where:

- F = annual probability of failure of fault category
- i = single component i in fault category
- n = number of components in fault category
- R_i = annual probability of no failure (i.e. reliability) of single component i
- N_i = total number of component i in WEC

The O&M model inputs can include array-based components as well as the components contained within an individual WEC. Where these aspects are included in the analysis, it is possible that the failure rates of related fault categories will change according to size of the overall array. For

example, the failure rate of the overalls moorings system might increase if more WECs are added to the array. Therefore, the probability of failure for array-based fault categories may differ from that calculated by equation A.5, for example:

$$F_A = 1 - (\prod_{i-1}^{n} Ri^{N_i})^{N_m} \qquad \text{(A.6)}$$

Where:

F_A = annual probability of failure of array-based fault category
i = single component i in fault category
n = number of components in fault category
R_i = annual probability of no failure (i.e. reliability) of single component i
N_i = total number of component i in single mooring system
N_m = total number of mooring systems in array (i.e. a function of the number of WECs)

The parameters associated with the repair (e.g. parts costs) of a fault category must be based on all of the components which are represented within that category. A bill of materials is a useful document to identify such parameters. As a consequence of this method, the parameters shown in the O&M model are averages of all the relevant components. If there is a large variation in any aspect then further breakdown of that fault category should be considered.