Appendix A. The *NavalTrans* Model

B.1. Overview

NavalTrans is an agent-based model, and it is developed on *Eclipse*, an open-source integrated development environment (IDE), using *Repast J* agent modelling toolkit. An instance of the model is composed of two main components; the *model code*, and the *parameter file*. The *model code* depicts the actions of the actors, relations between system elements, and general developments in the system. In other words, it depicts the structure of the model. The *parameter file* is a conventional MS Excel file, which contains the initial values of the model parameters that are needed to create an instance of the model. This architecture brings ease and flexibility to experimentation with *NavalTrans*: no programming skills are required in order to create different instances of the model and conduct experiments, since creating a new *parameter file* suffices for the task.

The model covers 2 naval transportation options (i.e. wind-powered sail-ships, and steam-powered steam-ships), and the behaviour of 9 actor groups. 7 of these groups are practitioner/user type of actors that constitute the demand-side of the transportation system. The remaining two actor groups represent the provision side of the system; i.e. provider-type actors.

In the following section, we will first introduce the *objects*, which are the basic pieces of the model. Following that, the behaviour of the overall model, as well as the *objects* will be discussed mainly using pseudocodes[1] and time-sequence diagrams[2]. When necessary, implemented functional relationships will be introduced in detail.

B.2. Object classes

Before directly going into the model discussion, some basic notions related to object-oriented programming, and naturally to agent-based modelling, are introduced below. *Objects* are the fundamental elements of an object-oriented program, and they can be considered as distinct pieces of software that have a state description, and a set of

---

[1] *Pseudocode* is a compact and informal high-level description of a computer programming algorithm that uses the structural conventions of a programming language, but is intended for human reading rather than machine reading.

[2] *Time-sequence diagrams* demonstrate the flow of actions that take place during a particular time-step (or an iteration) of the model run.

methods that depict the behaviour of the object. An *object* stores its state in fields (variables in some programming languages) and exposes its behaviour through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object interaction. In an agent-based model, which is a special type of object-oriented program, each individual agent in the model can be considered as an *object*, for example. However, there may be other objects that are not agents, which are needed for the functioning of the overall model.

A *class* corresponds to a general representation of a particular type of *object*. In other words, it is the blueprint from which individual objects are created in the program. Every *object* in the program is said to be an *instance* of a particular *class*. In an agent-base model, there may be numerous consumer-agents. Each of these agents is an *object* in the model, and the general description of a consumer is the consumer *class*.

In *NavalTrans* there are 6 object classes. 4 of these classes directly correspond to the concepts from the actor-option framework; *option*, *actor*, *practitioner*, and *provider classes*. The other two *classes* are more operational, and are related to the functioning of the overall model; *model* and *oracle classes*. We start with introducing these latter two *classes*, and then we will introduce the others.

a. *Controller* class

The *controller class* is the main controller of *NavalTrans*. First of all, the *controller* is responsible for creating an instance of *NavalTrans*. This involves initialization of the agents based on the parameter file being used, and setting the simulation run length. Secondly, the *controller* also controls the simulation schedule during a simulation run. The simulation schedule is related to the order in which different agents will be activated to perform their tasks in every simulation step. Finally, the *controller* class is also responsible for collecting data from agents during a simulation run, and displaying this data in the form of display surfaces and time-series plots.

There is only one instance of this class in *NavalTrans*.

b. *Oracle* class

As the name implies, the *oracle* class is related to information; it is the object that knows everything when *NavalTrans* is to be initialized. This class establishes the link between the *model file* and the *parameter file*. While initializing a particular instance of *NavalTrans*, the *oracle* reads the values of the model parameters from the parameter file, and reports these to the *model* object.

This class uses Apache POI HSSF libraries for reading data from MS Excel files.

There is only one instance of this class in *NavalTrans*.

c. *Option* class

The *option class* in *NavalTrans* corresponds to the naval transportation options considered in the case study. This class describes the way an option will be represented in the model (i.e. variables), as well as the mechanisms that alter the properties of the options (i.e. methods).

The basic set of variables of the option class is as follows;

Variable name	Short description
attrAct[i]	Actual level for the attribute i
attrBase[i]	Initial (base) level for the attribute i
attrBest[i]	Technically feasible best level for the attribute i
attrEconScale[i]	Whether the attribute i is influenced by economies-of-scale, or not (binary variable)
attrTechDevFracNorm[i]	Reference value of technical development fraction for the attribute i
attrType[i]	Type of the attribute i; indicates what type of development mechanism is active on the attribute i
capaTotal	Option's total vessel capacity
scale	Option's recent scale of utilization

The basic set of methods[3] of the option class is as follows;

Method name	Short description
attrUpdate(.)	Updates the attribute values of the option
updateStat(.)	Updates the statistics related to the option, such as utilization level, capacity-demand balance, etc.

There are two instances of the class in *NavalTrans*; sail-ship option, and steam-ship option.

d. *Actor* class

The actor class defines social actors in general. Independent of being a demand-side, or supply-side actor, the social actors share some basic characteristics in the way they are conceptualized for this model. The actor class corresponds to this general description of a social actor.

The basic set of variables of the actor class is as follows;

Variable name	Short description
attrPerc[opt][prop[4]]	The perceived level of the attribute *prop* of the option *opt*
attrPerDelay	Attribute perception delay of the actor
optScoreFunc	Component value function
prior[obj]	Priority of the objective *obj*
resource	Total amount of resources controlled by the actor
resToOpt[opt]	Resources allocated to the option opt
suppor[opt]	Support of the actor to option opt
supShiftNormal	Actor's reference rate of support shift

The basic set of methods of the actor class is as follows;

Method name	Short description
valueComponent(.)	Assigns an assessment score to an option with respect to a particular objective
valueDecision(.)	Assigns a total assessment score to an option

[3] The basic set includes the mechanism related to the behavior of the agents in the socio-technical context. The methods related to the functioning of the program, such as the ones related inter-object communication, data sorting, etc. are not discussed within the set of basic method for the sake of clarity

[4] i^{th} property of an option (i.e. *opt=i*) is related to the i^{th} objective (i.e. *obj=i*) in the preference structure of the actor

	considering the scores assigned with respect to all objectives
attrPercUpdate(.)	Updates the attribute values perceived by the actor

e. *Practitioner* class

The practitioner class is a subclass of the actor class. In other words, a practitioner is a special type of actor. It inherits all the variables and methods from the actor class, and on top of those it has some others that differentiate it from other subclasses. As the name implies, the practitioner class corresponds to the demand-side agents in the *NavalTrans* model.

The basic set of variables of the practitioner class is as follows;

Variable name	Short description
noSubType	Number of different groups of practitioners to be introduces
scaleShare[subType]	The share of each practitioner group in the transportation market
resGrowthPerc	Growth percentage for practitioner resources
resToProv[prov][opt]	Resources allocated to provider *prov* for transportation via option *opt*

The basic set of methods of the practitioner class is as follows;

Method name	Short description
supportUpdate(.)	Update the actor's support for different options
allocateRes(.)	Allocate resources to existing providers and options

There are 500 instances of the class in *NavalTrans*. In other words, there are 500 practitioner agents in the model. The distribution of these agents according to the sub-types is given below;

Practitioners/Users	*# of agents*
Merchant – Type I	50
Merchant – Type II	50
Merchant – Type III	50
Merchant – Type IV	50
Government Postal Service	50
Luxury Passengers	50
Emigrants	200

f. *Provider* class

Similar to the practitioner class, the provider class is also a subclass of the actor class, and inherits its variables and methods. This class corresponds to the supply-side agents in *NavalTrans*.

The basic set of variables of the provider class is as follows;

Variable name	Short description
capacity[opt]	Total option *opt* capacity controlled by the agent
capaInv[opt]	Capacity investment to option *opt*
capaLoss[opt]	Capacity loss in option *opt* due to depreciation
resFromPrac[pract][opt]	Resources received from practitioner *pract* related to

	option *opt*

The basic set of methods of the provider class is as follows;

Method name	Short description
supportUpdate(.)	Update the actor's support for different options
updateCapacity(.)	Update the total capacity controlled by the agent related to each option
updateCapaFree (.)	Update the unused freight capacity after processing the requests of the practitioner agents
updateResFromOpt(.)	Update resources received from practitioners for each option

There are 200 instances of the class in *NavalTrans*; i.e. 200 provider. The distribution of these agents according to the sub-types is given below;

Providers	***#of agents***
Individual Owners	150
Large Shipping Comp	50

B.3. Pseudocode of *NavalTrans*

The action flow in *NavalTrans* during a simulation run is presented by the following pseudocode. The individual actions mentioned in this pseudocode are explained in the following section.

```
Initialize the model instance {
        Read the parameter file for parameter values
        Create N_pract number of practitioners
        Create N_prov number of providers
        Create N_opt number of options
}
Repeat until the time step is equal to the simulation final time {
        Repeat until all practitioners are considered {
                Randomly pick one practitioner agent
                Ask the agent to perform the following {
                        Assess the available options (valueDecision)
                        Update perceived information (attrPercUpdate)
                        Update utilization of the options (supportUpdate)
                        Allocate resources to the options  (allocateRes)
                }
        }
        Repeat until all providers are considered {
                Randomly pick one provider agent
                Ask the agent to perform the following {
                        Assess the available options (valueDecision)
                        Update perceived information (attrPercUpdate)
                        Update the demand received (updateResFromOpt)
                        Update investment for the options (supportUpdate)
                }
        }
        Repeat until all options are considered {
                Randomly pick one option
                Ask the object to perform the following {
                        Update the option properties (attrUpdate)
                        Update the statistic (updateStat)
                }
        }
```

}
End simulation

B.4. Description of the agent actions

a. Practitioner actions

i. *valueDecision(.)*

Calculates a preliminary assessment value for all options by calculating a weighted average of its properties, as the actor knows them;

$$valueDecPrel_{opt} = \sum_{obj} prior_{obj} \times attrPerc_{opt,obj} \quad [B.01]$$

Then these preliminary assessment values are normalized based on the best option in the system according to the agent;

$$valueDecBest = \max_{opt}(valueDecPrel_{opt}) \quad [B.02]$$

$$valueDec_{opt} = \frac{valueDecPrel_{opt}}{valueDecBest} \quad [B.03]$$

ii. *attrPercUpdate(.)*

It is assumed that the learning takes place mainly through exogenous information sources, and the pace of learning is proportional to the imperfection in the actor's perception about a certain property of an option. According to this assumption, the perception of an actor regarding each property of each option is updated as follow;

$$attrPerc_{opt,prop}(step) = attrPerc_{opt,prop}(step-1) + correctionTerm(step) \quad [B.04]$$

$$correctionTerm(step) = \frac{attrAct_{opt,prop}(step) - attrPerc_{opt,prop}(step)}{attrPercDelay} \quad [B.05]$$

where *step* stands for the iteration step (or time-step) of the simulation.

iii. *supportUpdate(.)*

In NavalTrans, the *support* concept represents how much (in terms of percentages) of its freight a merchant wishes to transport via a certain option, or which portion of an emigrant group wishes to travel via a certain option, as already discusses in Chapter 9. Based on this interpretation, an increase in the support for an option means a decrease in the support of the other. The change in the actor's support for an option is formulated as follows;

$$\sup Shift_{opt1,opt2} = \begin{cases} A & \text{if } valueDec_{opt1} > valueDec_{opt2} \\ 0 & \text{otherwise} \end{cases} \quad [B.06]$$

$$A = (valueDec_{opt1} - valueDec_{opt2}) \times supShiftNorm \times support_{opt2} \quad [B.07]$$

$$support_{opt1}(step) = support_{opt1}(step-1) + supShift_{opt1,opt2} \quad [B.08]$$

iv. allocateRes(.)

Allocation of resources corresponds to allocation of the freight that needs to be transported to the available ships according to the current support levels of the actor (i.e. desired allocation of freight among transportation options), in the case of a merchant. In that respect, the workflow in this method is given in the following pseudocode;

```
Repeat for each option {
        Calculate the freight aimed to be transported using the option (resDes)
        Repeat until either all freight is assigned, or all providers are considered {
                Randomly select[5] a provider agent
                Ask for the free capacity controlled by the provider
                Assign freight to the provider according to the free capacity
                Ask the provider agent to update the free capacity it controls
                Update the amount of freight that needs to be transported
        }
        If there is still some unassigned freight, register it as surplus
}
```

If the *surplus* for an *option* is positive, this indicates that the actor could not find enough free capacity in order to realize the desired allocation of the freight among available options. For example, a merchant might have wanted to ship 50 tons via steam-ships. However, if the agent can only find free capacity for 30 tons on the steam-ships available in the market, then the *surplus* for steam-ship option will be 20 tons this practitioner agent. If this is the case, the agent tries to ship its freight through the other option. In this particular example, the merchant looks for free capacity of 20 tons on sail-ships.

```
Repeat for each option{
        Allocate the surplus to the other option
        If there is still some surplus, register the surplus as failed shipment
}
```

b. Provider actions

i. valueDecision(.)

The action is identical to the one discussed above for the *practitioner* agents.

ii. attrPercUpdate(.)

The action is identical to the one discussed above for the *practitioner* agents.

iii. updateResFromOpt(.)

The provider agent registers the total demand received from the practitioner agents for each type of ship the provider operates. For the provider with the index prov, the calculation is performed as follow;

$$resFromOpt_{opt} = \sum_{pract=1}^{N_{pract}} resToOpt_{pract,prov,opt} \quad for\ opt = \{sail,\ steam\} \qquad [B.09]$$

[5] The practitioner first randomly selects among the providers with whom it did business in the past, and then starts randomly picking out of remaining providers.

iv. supportUpdate(.)

In NavalTrans, the *support* concept represents how much (in terms of percentages) of its capacity investment resources a ship-owner wishes to allocate to a certain option (e.g. how many of the newly ordered ships will be sail-ships?). The formulation used for representing changes in this support allocation is identical to the formulation given in Equations B.06 through B.08.

Once the new support levels are determined, the provider agent calls for the *updateCapacity(.)* method. This method updates the shipping capacity controlled by the *provider* agent in both *options*. The method works as follow;

$$resTotal = \sum_{opt=\{sail,steam\}} resFromOpt_{opt} \quad [B.10]$$

$$capaInvTotal = capaInvFrac \times resTotal \quad [B.11]$$

repeat for each option{

$$capaLoss_{opt} = \frac{capaTotal_{opt}}{capaLife_{opt}} \quad [B.12]$$

$$capaInv_{opt} = capaInvTotal \times support_{opt} \quad [B.13]$$

$$capaTotal_{opt}(step) = capaTotal_{opt}(step-1) + capaInv_{opt} - capaLoss_{opt} \quad [B.14]$$

}

c. Option actions

i. attrUpdate(.)

The method updates the properties (i.e. attributes) of the options.

$$attrDevFrac_{prop} = attrDevFracNorm_{prop} \times eff_1 \times eff_2 \quad [B.15]$$

where eff_1 stands for the effect of investment flows on attribute development, and eff_2 stands for the effect of market-share loss on attribute development (related to *'fight-back'* or *'sail-ship'* effect)

$$eff_1 = f(\frac{capaInvFracAvg}{capaInvFracRef}) \quad [B.16]$$

capaInvFracAvg is the ratio of total capacity investments made for an *option* in a decision round to the existing total capacity of the *option* in the market. *capaInvFracRef* is the investment fraction that is required to compensate the capacity loss due to depreciations. The function *f(.)* is defined as follows;

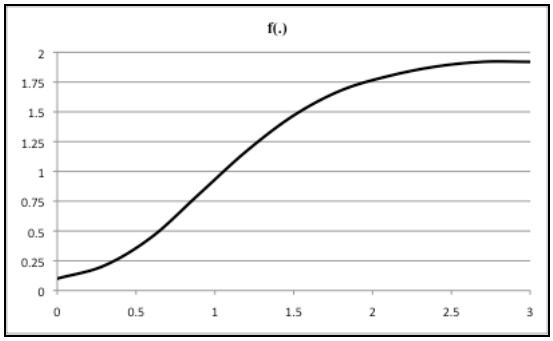

Figure B.1. The effect of investment flows on attribute development

$$eff_2 = g(\frac{supPractChgAvg}{supPractChgRef}) \qquad [B.17]$$

where *supPractChgAvg* is the average change in the support (i.e. demand) for the option in the market, and *supPractChgRef* is the reference value for that, which can be considered as normal. In the base version of the model a 2% market share loss is considered as acceptable (i.e. *supPractChgRef=-0.02*). The function *g(.)* is defined as follows;

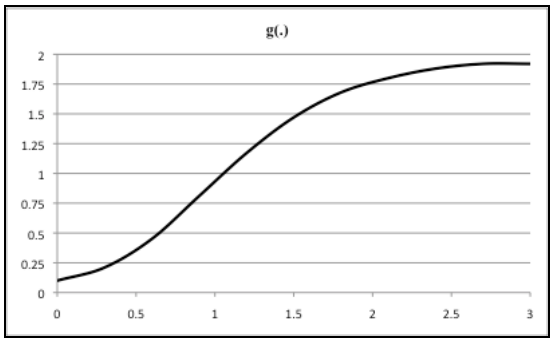

Figure B.2. The effect of market-share loss on attribute development

Using the *attrDevFrac* calculated in this way, the new level of the attribute is calculated as follows;

$$attr_{prop}(step) = \left(attrBest_{prop} - attr_{prop}(step-1)\right)attrDevFrac \qquad [B.18]$$

ii. <u>*updateStat(.)*</u>

This method calculates statistics about the whole market for the option by collecting demand-side data from individual practitioners, and supply-side data from individual providers. The method works as follows;

```
Repeat for every practitioner agent{
        Get the following information from the practitioner agent{
                Assessment score of the agent for the option
                The demand that the agent wanted to transport via the option
                The demand that the agent actually could transport via the option
                Total demand of the agent
        }
}

Repeat for every provider agent{
```

```
        Get the following information from the provider agent{
                Vessel capacity controlled by the agent
                Total investment resources of the agent
                Capacity investments of the agent for the option
        }
}
```

Using the collected information, the method calculates market-level indicators such as average assessment score for the option, average investment fraction to an option, demand-supply balance in the market, etc.

B.5. Parameter file

The parameter file consists of three sections corresponding to the *practitioner* agents, *provider* agents, and the *options*. Figure B.3 through B.5 are examples of these sections from a parameter file used for one of the experiments. A new *provider* or *practitioner* type can be added to the mode just by adding the corresponding rows to the parameter file. Same holds for introducing a new *option* to the model. *Disp_Factor_XX* columns in the *practitioner* and *provider* sections are related to the dispersion of the agents in the related issue. For example, a high *Disp_Factor_Res* indicates that the model will create agents highly heterogeneous in terms of the amount of resources they control. If the factor is set to 0, all agents will be initialized identical.

Attr Perc Sail	Reg	Speed	Opr. Cost	Inv. Cost	Range	Demand	Support
LR-LV Merchant	1	1	3	2	3	0	0.95
LR-HV Merchant	1	1	3	2	3	0	0.9
SR-LV Merchant	1	1	3	2	3	0	0.95
SR_HV Merchant	1	1	3	2	3	0	0.9
Mail	1	1	3	2	3	0	0.8
Luxury Passenger	1	1	3	2	3	0	0.85
Immigrants	1	1	3	2	3	0	0.99

Attr Perc Steam	Reg	Speed	Opr. Cost	Inv. Cost	Range	Demand	Support
LR-LV Merchant	3	2	1.5	1	1	0	0.05
LR-HV Merchant	3	2	1.5	1	1	0	0.01
SR-LV Merchant	3	2	1.5	1	1	0	0.05
SR_HV Merchant	3	2	1.5	1	1	0	0.01
Mail	3	2	1.5	1	1	0	0.2
Luxury Passenger	3	2	1.5	1	1	0	0.15
Immigrants	3	2	1.5	1	1	0	0.01

Total Numbers	Total No	Scale Share	Disp Fact Res	Disp Factor Attr	Disp Factor Prior	Disp Factor PercDel	Disp Factor Support	Attr PercDelay	SupShift Norm	ResPerCapa
LR-LV Merchant	50	0.05	0.3	0.25	0.25	0.2	0.3	40	0.015	1
LR-HV Merchant	50	0.3	0.3	0.25	0.25	0.2	0.3	40	0.015	1
SR-LV Merchant	50	0.05	0.3	0.25	0.25	0.2	0.3	40	0.015	1
SR_HV Merchant	50	0.15	0.3	0.25	0.25	0.2	0.3	40	0.015	1
Mail	50	0.15	0.3	0.25	0.25	0.2	0.3	40	0.015	2
Luxury Passenger	50	0.2	0.3	0.25	0.25	0.2	0.3	40	0.015	2
Immigrants	200	0.1	0.3	0.25	0.25	0.2	0.3	40	0.015	1.2
Total/Avg	**500**	**1**								**1.37**

Figure B.3. Section of the parameter file related to the practitioners

Priorities	Reg	Speed	Opr. Cost	Inv. Cost	Range	Demand
Shipowners	0	0	0	5	0	4
Shipping Comp	0	0	0	3	0	4

Attr Perc Sail	Reg	Speed	Opr. Cost	Inv. Cost	Range	Demand	Support
Shipowners	0	0	0	2	0	0.9	0.99
Shipping Comp	0	0	0	2	0	0.9	0.95

Attr Perc Steam	Reg	Speed	Opr. Cost	Inv. Cost	Range	Demand	Support
Shipowners	0	0	0	1	0	0.9	0.01
Shipping Comp	0	0	0	1	0	0.9	0.05

Total Numbers	Total No	Scale Share	Disp Fact Res	Disp Factor Attr	Disp Factor Prior	Disp Factor PercDel	Disp Factor Sup	Attr PercDelay	SupShift Norm	Capa Avg	Capa InvFrac	Capa UtilFrac
Ind. Shipowners	150	0.3	0.2	0.25	0.3	0.2	0.3	40	0.01	5000	0.06	0.8
Shipping Comp	50	0.7	0.2	0.25	0.3	0.2	0.3	40	0.01	2500	0.06	0.8
Total/Avg.	200	1										

Figure B.4. Section of the parameter file related to the providers

Base	Reg	Speed	Opr. Cost	Inv. Cost	Range	Demand
Sail-ship	0.66	0.66	2	1.33	2	0
Steam-ship	3	2	1.5	1	1	0

Best	Reg	Speed	Opr. Cost	Inv. Cost	Range	Demand
Sail-ship	0.66	0.66	3.5	1	4	0
Steam-ship	6	5	4	2	4	0

Dev Frac	Reg	Speed	Opr. Cost	Inv. Cost	Range	Demand
Sail-ship	0.005	0.005	0.005	0.005	0.005	0.005
Steam-ship	0.005	0.005	0.005	0.005	0.005	0.005

	Scale	CapaLife
Sail-ship	3700	20
Steam-ship	150	20
	3850	

	Reg	Speed	Opr. Cost	Inv. Cost	Range	Demand
Type	1	1	1	1	1	3
EconScale	0	0	1	1	0	0

Figure B.5. Section of the parameter file related to the options