Planktonica: A System for Doing Biological Oceanography by Computer

W. R. Hinsley

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Engineering of the University of London
and the Diploma of Imperial College

University of London
Imperial College of Science, Technology and Medicine
Department of Computing

March 2005

Abstract

One of the grand challenges in biological oceanography is the ability to predict the ocean plankton ecosystem using mathematical models. Most ecosystem models to date have been population-based and are defined by sets of coupled differential equations and solved as such. More recently the advent of high-powered computers has made individual-based models feasible, where agents represent individuals or sub-populations, each obeying a set of primitive equations or rules. Crucially, individual-based models have been shown to be stable over a wide range of parameters and avoid the chaotic instability exhibited by population-based models.

This thesis focuses on the software engineering issues associated with developing individual-based models of the plankton ecosystem. We present Planktonica, a problem-solving environment distinguishing the notions of *metamodel*, which abstracts the integration method and underlying physics, and *object model*, which defines a specific ecosystem model and chemical environment. We aim to reduce the modelling task to specifying the behaviour of individual organisms and their interactions with their ambient chemical environment using primitive rules. This enables complex models to be built by biological oceanographers, rather than by specialist programmers.

We describe the philosophy underlying Planktonica and document the implementation of a modelling framework that supports it. We develop a mathematical modelling language for describing primitive rules for biology and chemistry - essentially the language of mathematics augmented with functions that interface to the metamodel. The resulting system is evaluated by reconstructing existing individual-based models, and new models constructed entirely through Planktonica that demonstrate the system's ability to facilitate new science.

Acknowledgments

Special thanks to my supervisor Dr.Tony Field, for many hours of help, discussion, guidance and encouragement through this work, and to Professor John Woods for laying the foundations Planktonica builds on, and for strongly motivating and guiding the development of Planktonica towards meeting the needs of biological oceanographers. Also thanks to Adrian Rogers and Matteo Sinerchia for much time, effort, technical skill and encouragement along the way.

I also thank many people who have offered their time and expertise in many different ways through the course of this work: Professor Roger Wiley, Dr. Sarah Bennett, Dr. Paul Kelly, Dr. Steven Newhouse, Dr. Will Knottenbelt, Dr. Olav Beckmann, Dr. Angelo Perilli, Arun Rishi, Evan Weaver, Samir Al-Battram, and Rich Barber.

Finally, for personal encouragement, proof-reading and the provision of tea, I particularly thank Faye, James, Steve, my various families, the friends in my cell group and all at Every Nation Church London, and The Vine Church in Cranbrook. Dei Gratia.

Contents

List of Figures

Chapter 1

Introduction

This thesis is concerned with conducting mathematical simulations of plankton in a virtual ocean ecosystem. It contributes a modelling language for specifying the behaviour of plankton, and delivers suitable software tools for the creation and execution of computer simulations of models. These tools, combined with a set of applications from other sources, form the Virtual Ecology Workbench (VEW), a comprehensive suite of tools for creating simulations of biological oceanography, and analysing their results with the purpose of exposing the emergent properties of the models created.

1.1 Biological Oceanography

Biological oceanography is the study of flora and fauna in the ocean; this thesis is focussed on plankton, which are defined by Woods to be organisms that cannot usefully change their ambient environment by moving horizontally, although they may do so by moving vertically [56]. Figure 1.1, also reproduced from Woods [56], shows a hierarchy of the plankton community, in which plankton are separated into six classes: bacteria, viruses, phytoplankton, herbivores, carnivores, and higher predators such as fish larvae.

Many species and varieties of plankton exist, however many groups of plankton also exhibit the same basic physiology and behaviour. They are separated into *functional groups*, defined by a unique combination of phenotypic equations, where phenotypic

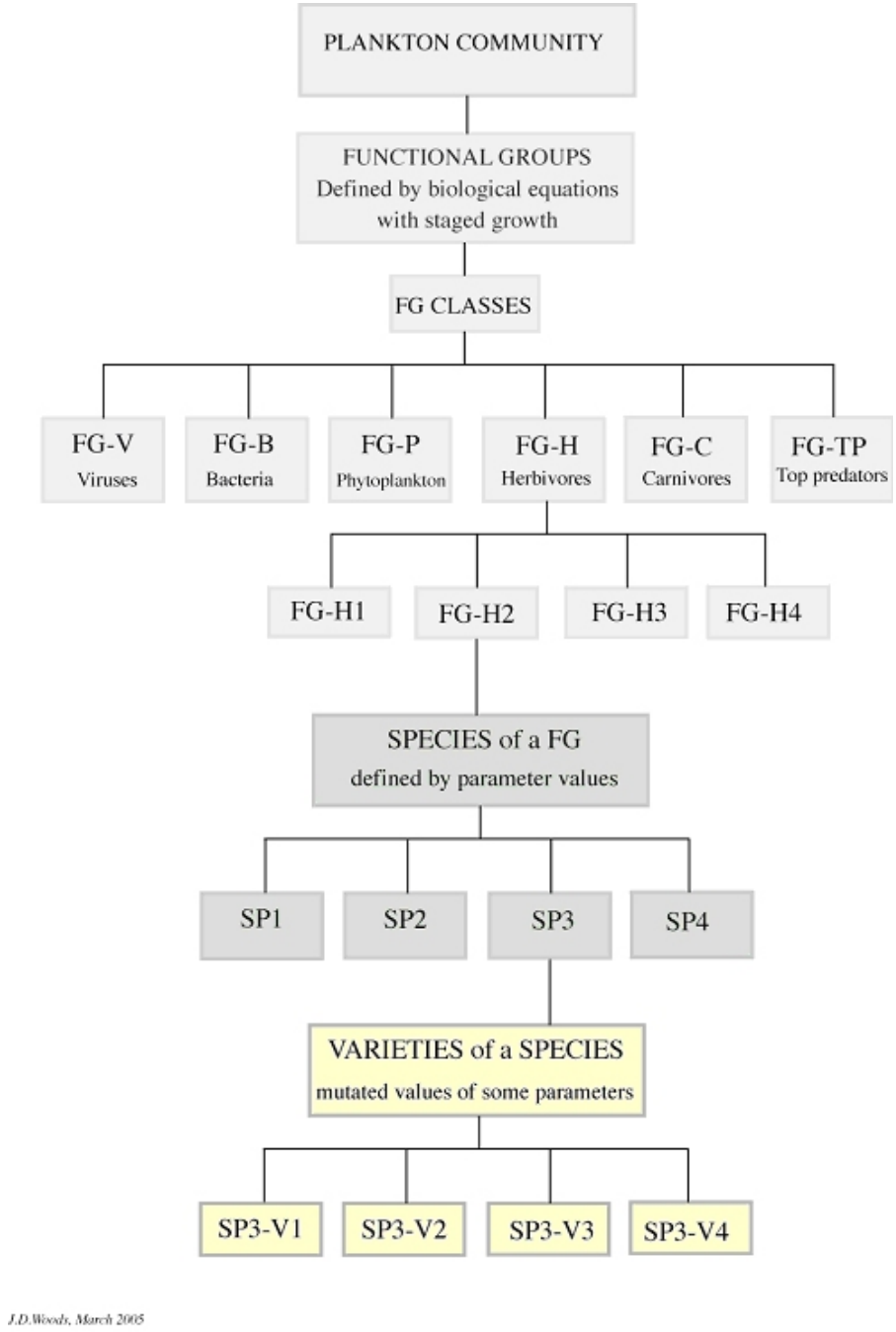

Figure 1.1: The Plankton Community (Woods, March 2005)

is defined as the observable physical or biochemical characteristics, as determined by genetic makeup and environmental influences. A species is a parameterisation of a functional group, and a variety is a specialisation of a species, defining a relatively small change to one or more parameters. Additionally plankton in a functional group may exhibit different behaviour in different stages of their growth, or in different seasons.

The ocean food-web defines the complex pattern of predation, where larger creatures feed on smaller, often with preferences for certain species and/or plankton of certain size. Phytoplankton are organisms that depend on sunlight and on nutrients to fuel growth by cell division. They perform photosynthesis, by which chlorophyll within the plankton affects the dissipation of light through the water. Thus, phytoplankton cause a bio-optical feedback on their environment. Biochemical feedback also arises from the transfers of chemical between plankton and their environment.

1.2 Motivation

More than half of the biological production on the planet is caused by plankton. Phytoplankton photosynthesis is responsible for about 70% of the oxygen production on the planet. Plankton play a significant role in determining the earth's climate, particularly in the regulation of atmospheric carbon dioxide. Understanding the beneficial and harmful effects of man's influence on the oceans [13], and the influence the oceans can have on man [45], is one of the most challenging open scientific problems.

For example, consider how the effect of certain industrial processes on the ocean ecosystem [13], or the spreading of harmful diseases such as cholera [45] could be predicted. In order to make such predictions, we must improve our understanding of the most significant end of the food chain: the population dynamics of plankton.

Investigations in biological oceanography can be undertaken by observation, or by mathematical simulation. Observation involves sampling at sea, producing data within the limits of instrumental and sampling error. However, such observations can only be done on a small scale, and the observer is required to know the context of the observations

being taken. For example, copepods swim downwards during the day and upwards at night, so measurements of copepod concentration must be made at an appropriate depth for the time of day. Additionally, the ocean ecosystem is complex enough that simple deduction from observations may not accurately or fully explain the processes being observed.

Mathematical simulation involves building models of processes in the ocean, using computers to integrate the model over time. Various types of modelling approaches (or *metamodels*) exist, some of which involve writing equations to describe whole populations, others involve rules that describe the behaviour of individual plants and animals. Creating such models is challenging, since our understanding of the ocean is incomplete. However, we do have a good understanding of some biological processes, including the primitive behaviour of individual plankters, based upon laboratory experiments.

With this information, it is possible to create a virtual environment with customised conditions, to simulate a number of plankton, and to observe the emergent properties of the plankton population over time. This enables any number of hypothetical scenarios to be created. Of course, the validity of results obtained from such experiments depends on the model of plankton behaviour being correct.

This thesis focuses on the creation of mathematical simulations of ocean life, using computer science to allow biological oceanographers to build models of the ocean ecosystem.

1.3 Simulation

The biological study of plankton, both plant and animal has a very long history, and the physiology and behaviour of many individual organisms are well understood. However, while carefully controlled laboratory experiments can be used to observe the dynamics of an individual, an entire population cannot be observed in such a way. The only way to understand the complex interactions between individual, population, and environment is to construct a mathematical model of each and to solve that model numerically. A

scientist's analysis of such a model may yield explanations of any observed behaviour such as seasonal blooming, toxic bloom formation, population distribution or chemical transportation.

Furthermore, simulation facilitates scenario prediction, which explores the ecological consequences of changes to exogenous conditions. Examples include the response to artificial fertilisation of the oceans, the effects of climate change and the dispersal of pollutant chemicals or diseases.

Population-based simulations treat populations of a functional group as a single entity. They use prognostic equations to compute demography and biofeedback directly. The Fasham model [14] is the definitive population-based plankton model. When compared to individual-based modelling, population-based models have the advantage that they are extremely computationally efficient, as they require numerical solution of a modest number of coupled differential equations. Until the advent of powerful microprocessors and supercomputers, population-based modelling was the only feasible method of simulating plankton ecology. Population-based modelling has the disadvantage of exhibiting instability for certain choices of parameter values [40, 59, 30].

The opposite extreme is pure individual-based simulation, in which every individual is treated as an independent entity, with rules written for the individual rather than for the population. This gives the virtual ecosystem the freedom to 'adjust gracefully to changes in exogenous forcing' [56]. However, conducting such models can become prohibitively computationally expensive when very large numbers of individuals are to be simulated.

1.4 The Lagrangian Ensemble Metamodel

The Lagrangian Ensemble (LE) Metamodel [56] is a compromise between the population-based and individual-based approaches. It treats each particle in the simulation as a carrier of a number of identical individuals, called a sub-population. The size of the sub-population is dynamic during a simulation and the number of sub-populations

is controlled by adjustable particle management rules. Thus, if the number of sub-populations is set to one, the LE 'metamodel' can be used to emulate a population-based model, whereas if all sub-population sizes are one, then it emulates some of the characteristics of an individual-based model. Note that the LE method does not permit direct interaction between individuals (see section 3), so there is some distinction in the latter case.

The LE Method provides a method of addressing computational performance by limiting the number of individual particles to be simulated, while also providing an adjustable control over demographic noise.

1.5 Automation

Building computer versions of ocean models in conventional programming languages is an error-prone and complex process that requires software engineering skills generally beyond that of the typical biological oceanographer. The main objective of this thesis is the design, implementation and evaluation of a problem solving environment that provides tools that a biological oceanographer can use to build and analyse models, without the aid of an experienced software engineer. Hence, the software engineering investment is made 'upstream', in order to reduce the need for skilled programmers 'downstream'.

The aim is to reduce the task of building a model to that of writing rules for the biological functions of individual plankters, which is the way that biological oceanographers often think about the individuals being modelled. This is done by the introduction of a modelling language that is largely based on mathematical equations, but additionally carries a small number of ecosystem-specific functions for supporting the LE metamodel. This is intended to be easily grasped by biological oceanographers, empowering them to build LE simulations rapidly and analyse the models' emergent behaviours.

1.6 The Virtual Ecology Workbench

The Virtual Ecology Workbench (VEW) is a suite of applications designed to facilitate the building of Lagrangian Ensemble-based models, and the analysis of their emergent properties. The first version of the VEW did not achieve this goal: it relied on commercial software for model building which did not solve the problem of making model-building accessible to end users. As a result, models were mostly hand-coded in C by computer scientists.

The second version added interfaces for varying parameter values, creating scenarios for models and managing job control, but this still relied on underlying model code hand-crafted in C; it did not introduce methods for creating new models.

This thesis contributes the 'engine' to the third version of the VEW, which for the first time offers complete automation of Lagrangian Ensemble based simulations. It incorporates a completely new architecture for model building, using a model specification file that is created by a model builder with an equation editor interface, extended by a number of other applications and compiled into an executable simulation instance in Java.

This version of the VEW has been built by a team of researchers. The specification for the VEW has been developed by Professor John Woods, and the engineering of the software has been supervised by Dr. Tony Field. This thesis presents Planktonica: a problem-solving environment consisting of various user interfaces that expose to the user an underlying modelling language, allowing them to create and edit new models of plankton. Planktonica includes a compiler that produces executable Java classes from models, and a prototype of an interactive debugger.

Other applications contribute to the specification of a model: the software that guides the user through specifying the scenario, parameter settings, job control and post-simulation analysis was developed by Adrian Rogers. Additional work on viewing climatological data was done by Matteo Sinerchia, who is using VEW 3 to develop new models for testing theories of fisheries recruitment. See chapter 7 for more information

about the components of the VEW.

1.7 Planktonica

Planktonica encapsulates a formal implementation of the Lagrangian Ensemble metamodel. The metamodel describes the kind of simulations that can be built, including the definition of a particle, the ways that particles may interact, and the environment in which the particles exist. In particular, the metamodel defines the environment as a 1-D water column stratified into layers, where each particle's position is defined by its depth. Each particle is a sub-population of individuals, but when building the model, the user does not have access to the particle's sub-population size, or to the internal properties of any other particle.

A separation is thus introduced between the metamodel and the model, which can be thought of as a *curtain*. The metamodel properties described above exist behind the curtain, and are the basic rules for construction of all simulations. In front of the curtain, Planktonica provides a mathematical modelling language for designing plankton. Arbitrary chemicals can be introduced, which may have action spectra to represent pigmentation.

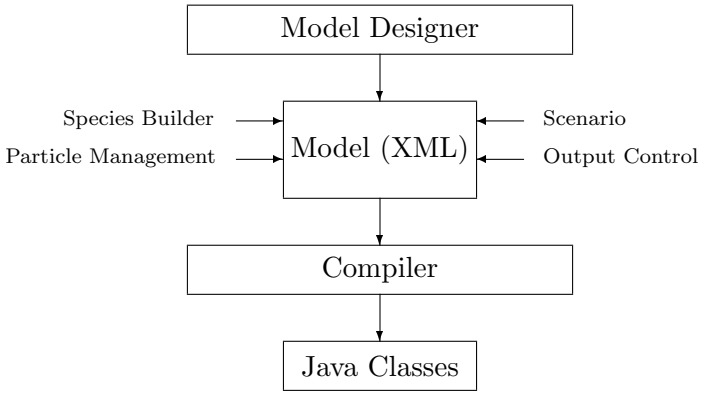

Figure 1.2: The Role of Planktonica

Figure 1.2 shows the role of Planktonica. The *Model Designer* is a graphical user

interface exposing the mathematical modelling language to the user, and the creation of a set of functional groups, chemicals, and associated rules consistent with the requirements of the Lagrangian Ensemble metamodel. The newly created model specification is represented in an XML document, which is then modified by the other VEW applications mentioned above. Finally, Planktonica's compiler produces Java classes which form an executable simulation.

1.8 Objectives

The over-riding objective is to substantially raise the level of abstraction used in the model development process so that modelling becomes a process of composing clearly identifiable and potentially re-usable model components. Crucially, the components and the model structure should make sense to a modeller. Three major questions arise:-

- What abstractions, modelling formalisms, development tools etc. should be provided to enable a biological oceanographer to build a new model, without resorting to low-level 'programming' in the traditional sense?
- To what extent can a component framework based exclusively on primitive rules or primitive equations be used to construct scientifically useful ecosystem models?
- To what extent does an integrated framework that supports a rule-based LE approach simplify the task of model design, implementation and analysis.

The rest of this thesis can be seen in part as an attempt to answer these three questions.

1.8.1 Challenges

Separation of Concerns

The first challenge is to introduce a separation between the concept of *metamodel* from *object model*. In the past, models have been implemented as sets of C modules, with

metamodel and object model combined. This made the implementation of new extensions difficult, through code complexity, and the lack of opportunity for re-use of components. This gave rise to a legacy problem in which many different versions of an underlying reference model, each with a different set of modifications, came into being.

This requires identifying every point where there must be communication between metamodel and object model, and providing a suitable interface. Since the target user is not a software engineer, the interfaces between metamodel and object model must be as few, and as simple as possible.

Model Formalism

A modelling formalism, and a suitable tool set is required to allow rules to be built in a 'programmerless' way. Of course, any notation can be considered a programming language; the objective is to constrain the semantics of that language in order to strike an acceptable balance between simplicity and expressibility. Experimental biologists and biological oceanographers use the language of mathematics to describe primitive biological processes, often in the form of primitive equations or rules. A suitable formalism should be in a similar style.

Internal Consistency

Much of the complexity of model development is in ensuring that the model is internally consistent. Rules are built out of mathematical constructs, using different types of variables and parameters, which have associated units. Mistakes involving units are common, for example interchanging chemical concentrations between micrograms and millimols.

While those errors might be considered as basic, they are common and often missed. Having made the modelling language familiar to biologists, the tools should also aim to address common problems with designing such models, such as type checking and unit checking.

Evaluation

This work is interdisciplinary in nature and sits at the boundaries of conventional computer science and biological oceanography. This thesis does not specifically aim to present novel computer science, nor to develop a deeper understanding of the plankton ecosystem per se. The objective is rather to apply sound computing principles to the development of tools that enable the end users to undertake novel and interesting science as efficiently as possible.

Specifically, the challenge is to provide software tools that enable biological oceanographers to build and analyse models of ocean life, and success is thus measured by the ease with which they may do so, rather than, necessarily, the novelty of the science that results from any particular case study.

The initial milestone for Planktonica's development is the re-implementation of the WB model [4], a Lagrangian Ensemble-based model already in use [57, 54].

Two further variations are then introduced. The first models the deposition of a buoyant, light-absorbing pollutant, that disperses over time, and has an occlusion effect. Analysis shows the effect of dispersion rate of the pollutant on the turbidity of the water, and the corresponding effect on the population of phytoplankton.

The second experiment, based on previous work by Simon Smith [47], introduces a delay between ingestion and excretion for copepods. A pollutant is added which diatom absorb. Copepods ingest the diatoms, excreting chemicals they gained by the ingestion a while later. The time between ingestion and excretion affects the rate at which the inert chemical is transported, as shown by this experiment.

1.8.2 Contributions

The contributions of the thesis can be summarised as follows.

- A metamodel kernel is presented, incorporating a physical environment supporting optics, turbulence and biofeedback. A chemical environment is included allowing arbitrary chemicals and pigments to be included.

- A mathematical modelling language, with accompanying operational semantics, is introduced, enabling users to describe the behaviour of plankton. It includes a small number of special purpose functions that allow interactions between object model and metamodel.

- A new Planktonica-compliant rendering of the WB model is presented in full, which can be used as a base for creating a range of new scientific experiments, as demonstrated.

- The modelling language, accompanied with various prototype interfaces, forms a platform independent problem solving environment for 'programmerless' creation of models, promoting modularity, automatic documentation, and a prototype interactive model debugger. This is presented in detail.

1.9 Contents of this Thesis

Chapter 2 describes the background to plankton research, beginning with a summary of population and individual-based modelling, and particularly the Lagrangian Ensemble method (section 2.2.1). We also outline the history of the Virtual Ecology Workbench, on which this work builds, and compare the simulations to agent-based artificial life simulations.

Chapter 3 focuses on the distinction between metamodels and object models, beginning by defining in more detail what metamodels are. This addresses the separation of concerns challenge, using the concept of a *curtain*. Elements in front of the curtain are accessible to the user; behind the curtain there are metamodel properties regarding the Lagrangian Ensemble method and the virtual physical environment. The choice of where to place the curtain (i.e., what to expose to the user), is explained.

Chapter 4 defines in general the type of rules that can be built by the user. Our discussion is informal at this stage. A formal operational semantics for the modelling language is presented in chapter 5. These two chapters are then illustrated in chapter 6, with a thorough description of the WB model, as re-implemented using Planktonica.

The model builder fits into the Virtual Ecology Workbench (VEW), a suite of applications that provide the required information to turn the model into a virtual ecosystem. This includes the specification of species and varieties, the scenario, initial and boundary conditions, particle management, output options and run control. Chapter 7 outlines the bigger picture of virtual plankton ecology and the way in which Planktonica contributes to the larger process from initial design to final analysis. Of these, the final compilation stage (section 7.8), and LiveSim, the interactive simulation viewer (section 7.10) are delivered as part of this work. The other parts are contributed from other researchers, as we describe.

Planktonica is then evaluated in chapter 8 by demonstrating how it can be used to facilitate new science. Results are presented from the new implementation of the WB model, and using that as a base, some simple experimental models are presented that demonstrate Planktonica's potential as a research tool.

Chapter 2

Background

There are broadly two approaches to modelling ecosystems. *Population-based* models consist of differential equations defining the change in a population size over time. Such models are solved directly by numerical integration. *Individual-based* models consist of equations that describe the behaviour of an individual; a simulation consists of a number of autonomous individuals. These models are solved by computing sample trajectories of each individual over time. In order to reduce demographic noise, a large number of individuals may be required; this incurs a computational expense that has historically limited the use of individual-based models. Population-based models are computationally cheaper, since they describe the plankton population as a continuum rather than as individuals.

2.1 Population-based Ecosystem Modelling

One of the most widely used models of the plankton ecosystem is the Fasham model [14]. This is a population-based model consisting of seven coupled differential equations that describe the evolution over time of phytoplankton, zooplankton, bacteria, detritus and particulate organic nitrogen, nitrate, ammonium and dissolved organic nitrogen. The model equations are coupled by the flow of nitrogen between compartments, where the biological and physical processes causing transfer of nitrogen include grazing, nutrient

uptake and mixing. Each of the seven principal equations is defined as the rate at which that population or chemical concentration may increase or decrease with respect to time, in terms of the behaviour of the other components; this includes zooplankton grazing, phytoplankton growth by photosynthesis, and detrital remineralisation.

Around thirty parameters are used. Ten of these are parameters are adjusted, sometimes to fine-tune the model and sometimes because no obvious estimate is available for that parameter. The relative simplicity of the model compared to other comparable plankton models [26], has made it a popular platform. ERSEM [3] is another example of a more complex plankton ecosystem model using population-based equations.

2.1.1 Stability

One of the key problems with population-based models is that they are unstable for some parameterisations - an artefact of the model itself rather than the nature of the integration method used. In the context of the Fasham model, Popova, Fasham et al [40] suggest that the instability shown represents inherent instability in the plankton ecosystem itself. In contrast Woods et al [59] argue that the instability is due to population-based handling of biofeedback, and that the real plankton ecosystem is inherently stable, and possible to predict via an individual-based approach.

A further complication is known as the 'closure' problem. The highest trophic level specified in the Fasham model is the Zooplankton herbivores, but in practice, the zooplankton population is limited by higher predators not specified in the model, and the population of the higher predators is not available. This causes a problem calculating the amount of such predation that should occur. The choice of closure equation was shown by Steele and Henderson [48] to significantly affect the dynamics of simple models, supported and formalised more recently by Edwards and Yool [12].

2.2 Individual-based Ecosystem Modelling

Individual-based modelling (IBM) uses rules or equations written for individual particles, rather than for the whole population. As the state and trajectory of each individual must be modelled explicitly, IBM has historically been restricted by computational cost, but advances in computer performance now make it more tractable.

Given definitions for what an individual in the simulation does, IBMs aim to observe and explain the *emergent properties* of a system: the histories of the particles, the demography of the plankton, and the resulting feedback to the environment. Since the particles contribute individually to this feedback, their combined effect is hard to predict, except by these kind of simulations; this is true of plankton ecosystems both virtual and real. Diagnosis of the results of individual-based models is shown to produce stable results more consistent with observation than those of population-based results [59].

Individual based models have been used to expose the emergent properties of many ecosystems, for example cockroaches [35, 1], birds [41], foxes and rabbits [18], and bees [49]. However the plankton ecosystem presents a more difficult challenge because of the microscopic scale of the organisms, and the enormous number of individuals required to build realistic scientific models.

2.2.1 The Lagrangian Ensemble Method of Integration

The Lagrangian Ensemble method [56], introduced by Woods and Onken [57, 54] is a specialisation of an individual-based modelling system, and a compromise between the need for individual-based modelling, and the computational cost when applied to microscopic life. Rules are written for an individual plankter, but using the LE method, each agent represents a set of identical plankters. These plankters have the same state, represented by a number of internal variables which changes according to the rules, in response to the ambient (i.e. local) environment. The population then has a feedback effect on the environment, which in turn affects the ambient environment of the individuals at their next location.

In such a system, the use of sub-populations allows for a quantitative compromise between reducing noise and improving performance, while the biofeedback mechanism proves to be critical for stability [59].

The WB model [4] is an individual-based model representing a plankton ecosystem. It is similar to the Fasham model in terms of the functional groups and chemicals that are modelled, but differs in that the rules are specified for individuals rather than for whole populations. Furthermore the rules for the plankton in the WB model are derived from laboratory experiments, in contrast to most population-based models, where parameter values are often guessed.

The model is integrated using the LE method, where each individual in the simulation represents a sub-population of identical plankters with the same state properties, but separate sub-populations have separate states and trajectories. A comparable method has also been used by Broekhuizen [7]. In their original form, both the WB model and Broekhuizen's model were represented as modules of C-code, hand-crafted by a programmer. We believe that Planktonica represents the first attempt to make model building a task that biological oceanographers can use directly without hand-coding in a conventional programming language.

2.3 Modelling Languages and PSEs

A problem-solving environment or PSE is defined as a 'computer system that provides all the computational facilities necessary to solve a target class of problems' [16]. PSEs aim to exploit hardware and algorithm power to deliver 'cheap and fast' problem solutions with minimum knowledge in computer programming [20]. They are not entirely programmer-less, since any simulation requires representation in some language, however PSEs aim to make that language as natural to the user as possible, rather than using languages common to computer science.

Within the realm of mathematics, MatLab [29] and Mathematica [28] can be thought of as PSEs. They provide ways of writing differential equations and interactively exam-

ining their results; Netsolve [2] additionally provides the interfacing to network resources for efficient or parallelised solutions of large problems. These systems provide a range of functions useful within their target problem class, in the case of Netsolve the efficient numerical solution of differential equations.

More often, systems like Wise [25] allow the user to link together a number of pre-written modules while specifying couplings between the modules, and different initial conditions or parameter values. Wise is targeted at ecosystem modellers, particularly involving nutrients in soil. A similar system for plankton research is the predecessor to this work, the Virtual Ecology Workbench 2 [55], which allowed grouping and parameterisation of pre-written C modules for different plankton types and nutrients. In both examples, the creation of new modules is not supported, except by explicitly programming them.

2.4 Agents and Artificial Life

Particles in a plankton simulation can be thought of as agents: individuals following a set of rules. The rules for agents may either be fixed, i.e. non-adaptive, or they may adapt over the course of a simulation [11]. Agent-based simulations have been written in a 'one-off' way for many specific domains, providing users with the ability to change parameter values for the particles and observe the results [38, 41, 18, 10]. In such simulations, the behaviour is usually fixed into the simulator, and parameterised by the user.

Agents have been designed that represent living creatures, either observed or imaginary; computer representations of the creatures and their behaviour are termed *artificial life* [34]. Artificial life-forms exist in many forms; Tamagochi and similar toys [9] were popular in the 1990s, while 'Creatures' [23] were the equivalent of Tamagochi on desktop computers. DaliWorld [6] allowed users to create a fish on their desktop that would swim to other DaliWorld users' desktops, thus forming a virtual distributed world-wide ocean; but these artificial life projects were more about creating new animals than modelling

existing ones.

The algorithm of Boids [41], originally a simulation of birds flocking, has been widely developed in films to model steering and obstacle avoidance of flocks of creatures; it was used to model swarms of bats and an army of penguins in Batman Returns, a gallamunus herd in Jurassic park, and a wildebeest stampede in The Lion King [39].

Frameworks also exist for creating new agent-based simulations. Many of these like Swarm [24] or Echo [21] take the form of programming libraries, providing computer scientists with convenient tools to write models quickly, simplifying mundane tasks like memory management or file handling, and providing various functions for particle interactions and so on. However, they rely on programming skill in Java or C for new simulations to be built using the libraries.

Higher level systems are also being developed. For example Adise [15] allows a user to design adaptive agents using mathematical rules via a high-level user interface, using plug-in modules for the environment. Destiny Studio [50] is a recent high-level simulation environment for building agent-based models without the need for conventional computer programming. NetLogo [51] uses an extended version of the Logo language to support agents and concurrency, providing a rich user interface for designing rules.

The strengths of these environments are in their rule-building capabilities. We aim to do the same here, but for the Lagrangian Ensemble method of integration. The aim is to provide flexibility - being able to create a wide range of rules - while also keeping model-building simple.

2.5 Virtual Ecology

The Virtual Ecology Workbench (VEW) is a problem-solving environment specifically for the plankton ecosystem, first developed by Woods and Barkmann [4], incorporating the Lagrangian Ensemble method of integration [57]. The central aim of the VEW is to allow design, simulation, and analysis of plankton models to be done in one problem-solving environment.

A model in the VEW consists of a water column divided into one-metre layers. The surface area is somewhat arbitrary - typically it is in the range of a metre to a kilometre squared, depending upon the application. The simulation reads forcing climatological data and uses it to calculate the irradiance, temperature, density and salinity throughout the column. It is also used to calculate the turbocline, above which the water is assumed to be randomly mixed and below which there is assumed to be laminar flow.

The biological particles in the WB model are phytoplankton which are plant-based plankton that gain energy by photosynthesis and zooplankton, which are herbivores that ingest the phytoplankton. Phytoplankton absorb nutrients including ammonium and nitrate from the water, and they also contain chlorophyll which has a biofeedback effect on the irradiance and temperature of the water. The WB model is essentially an individual-based equivalent of Fasham's original population-based model.

VEW 1 assembles C code to represent this behaviour, by importing ready-made modules of code from a library, offering tools to alter the parameters, initial conditions, and boundary conditions for the simulation, and to analyse the results afterwards. It cannot, however, create new particles, or add new rules to existing particles. Furthermore due to the difficulties in compiling the assembled modules, users with programming knowledge typically carry out edits on the final code rather than at the interface level. Thus many different instances of the base code exist, many containing different improvements or corrections. This has given rise to a code legacy problem that historically has complicated the development of new models.

2.6 Summary of Planktonica

This thesis is about creating a new Virtual Ecology Workbench, allowing for the first time the creation of new plankton models without needing to hard-craft them in conventional programming languages. Simulations are individual-based, incorporating the Lagrangian Ensemble method of integration. Each individual is a Lagrangian-Ensemble based agent, with rules describing its behaviour. The WB model is used as an example

of a plankton ecosystem model.

Chapter 3

Modelling Ecosystems

This chapter contains a description of the fundamental components of a model ecosystem, within the proposed framework. It begins by defining the *metamodel* - the fixed parts of the simulation - and the *object model*, which defines the biology and chemistry of a particular experiment. Interfaces between the two are required, and are described in this chapter.

3.1 The metamodel

The *metamodel* specifies all the generic properties of *object models* that can be built using that metamodel; it is the framework that all object models must fit. The metamodel is revealed primarily by a user interface, permitting the user to create only models that fit the generic properties of the metamodel. Accompanying this interface is a compiler that produces new Java classes from the object model, which along with some pre-defined Java classes form an executable simulation. The role of the metamodel and object model is shown in figure 3.1.

The divide between the metamodel and the object model can be thought of as a curtain separating what is visible to the user from what is not. The aim of this separation is to reduce the task of constructing a model to that of defining the biology and chemistry; all generic behaviour and implementation details are hidden *behind the*

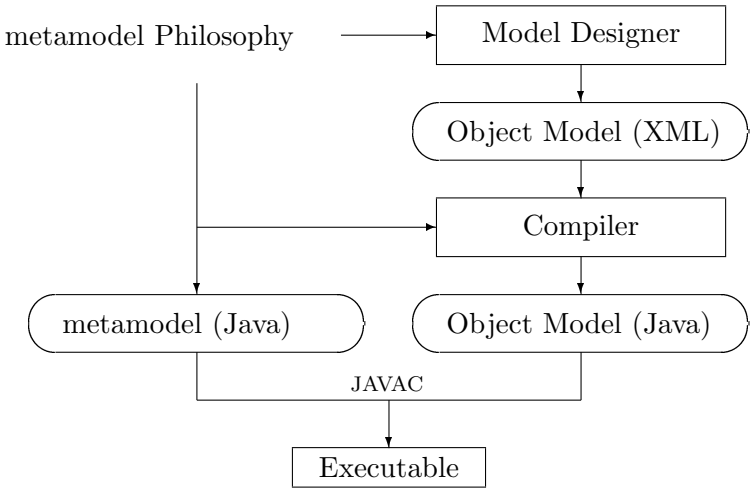

Figure 3.1: Metamodel and object model

curtain, whereas the particles and chemicals are designed by the user *in front of the curtain*. This terminology will be used throughout this chapter.

When the user needs to refer to properties behind the curtain, special interfacing methods are provided, either though exposing read-only variables for the user to incorporate into rules, or by calling special purpose functions from a library (i.e., an application program interface, API).

3.2 The Community

Figure 3.2 shows the hierarchy of groups of plankton represented by Planktonica. The challenge here is to allow the user to easily create many different kinds of plankton. In nature we observe that many different species of plankton exhibit the same basic behaviour, but at different rates, dependent perhaps on the characteristics of that species, their size for example. Therefore, we separate the behaviour of plankton from the characteristics that affect their behaviour.

A functional group contains the rules for the behaviour of a group of plankton. This includes the option to define stages for a functional group's life cycle, where it may

behave differently in each stage.

By default, a single species of each functional group, and a single variety of each species are created by Planktonica, meaning that all members of the functional group will behave in exactly the same way. The user specifies biodiversity by creating additional species and varieties, which are new parameterisations of that functional group.

A species in Planktonica defines some property that affects its behaviour. For example, we may create a species of plankton that behaves in a way related to its size (allometry), and another species that behaves proportionally to its maximum swimming speed. We call this dependent property the *base parameter*.

Having created a species with a base parameter, the user can then create varieties; each is a group of plankton with the base parameter set to a certain value. Thus the user can quickly create a number of parameterisations of a functional group to represent biodiversity. All particles in the simulation exist at the variety level. See section 3.4.3 for further detail about how biodiversity is represented.

3.2.1 The Lagrangian Ensemble Method

The Lagrangian Ensemble Method considers sub-populations of individuals; each sub-population is represented as a particle in the simulation, which is defined at the variety level of the plankton community. Users specify biological behaviour in terms of individual organisms; the sub-population size is not made visible to the user, and cannot be directly changed. Special functions are required to describe behaviour in which the sub-population size is relevant.

The model is integrated in timesteps. In each timestep the state of each particle is updated. This may result in a change in the internal state of the particle, including its sub-population. The timestep size is exposed to the user as a system variable $\triangle t$, as it will be required in all rules that describe a change over real time.

Particle-to-particle interaction is forbidden; only particle-to-concentration interaction is allowed. A particle may prey or graze upon another only by interacting with the target concentration. The number of individuals of a given type (its concentration)

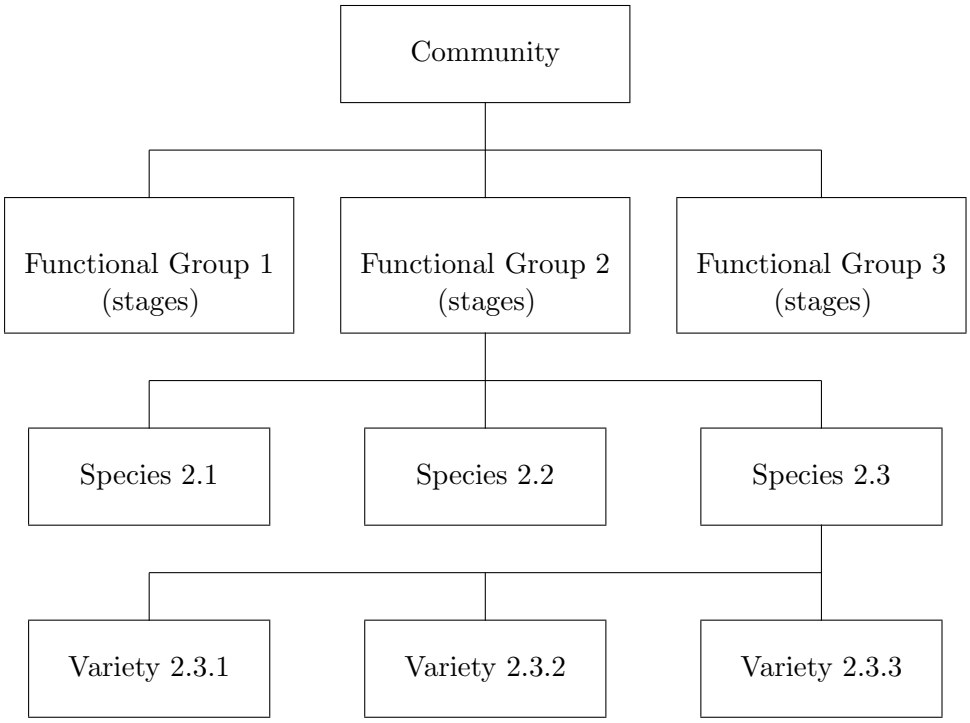

Figure 3.2: The Plankton Community

is computed automatically each timestep, in each layer of the water column, defined below.

3.3 The Environment

3.3.1 Physics

In the Planktonica metamodel, the ocean environment is represented as a water column stratified into layers, (see figure 3.3) making it a one-dimensional grid. At each grid point, values are maintained for the temperature ($^{\circ}C$), density (kgm^{-3}), salinity and irradiance (Wm^{-2}). Irradiance is available in two forms; I_V, visible irradiance, defines the energy due to irradiance for just visible wavelengths, whereas I_F, full irradiance,

defines the energy due to irradiance including the infra-red and ultra-violet wavebands. These two types of irradiance are used internally within the physics, but are both exposed to the user, along with temperature, density and salinity, for writing rules.

The water column may be fixed to a certain geographical location, or it may drift in response to ocean currents and wind. In either case, forcing climatological data is read in for each timestep, and is used to calculate the properties of the column at each grid point, and also the depth of the mixing layer, exposed to the user as the variable *MLDepth*. Above this depth, turbulent mixing occurs, and below this depth is laminar flow. The physics of the water column is predefined and cannot be changed by the user.

Internally, the grid points for physical properties are not equally spaced: higher resolution is provided near the surface, but this is unimportant from the users' perspective.

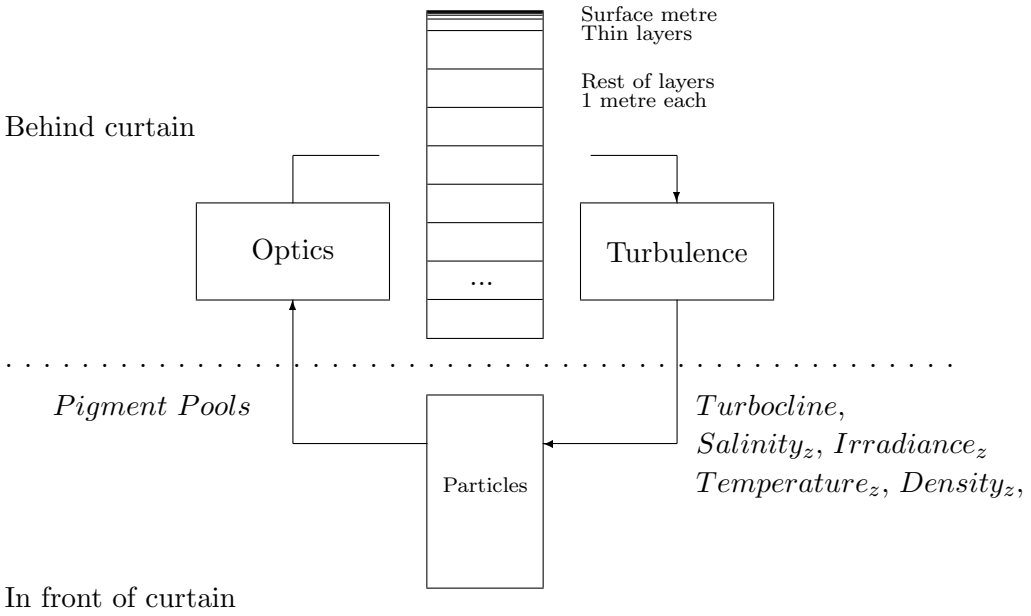

Figure 3.3: The Physical Environment

3.3.2 Chemistry

The environment supports arbitrary chemicals specified by the user. Each chemical is modelled as a continuum; the introduction of a chemical c, shown in figure 3.4, automatically introduces a variable c_{conc} at each grid point, and two variables within the particle: c_{pool}, the internal pool and c_{uptake} which stores the amount of chemical gained by the particle through uptake, in the previous timestep. The c_{uptake} variable is calculated by the metamodel in each timestep, and has a special purpose when handling chemical conservation, described below. All of these automatically created variables are visible to the user.

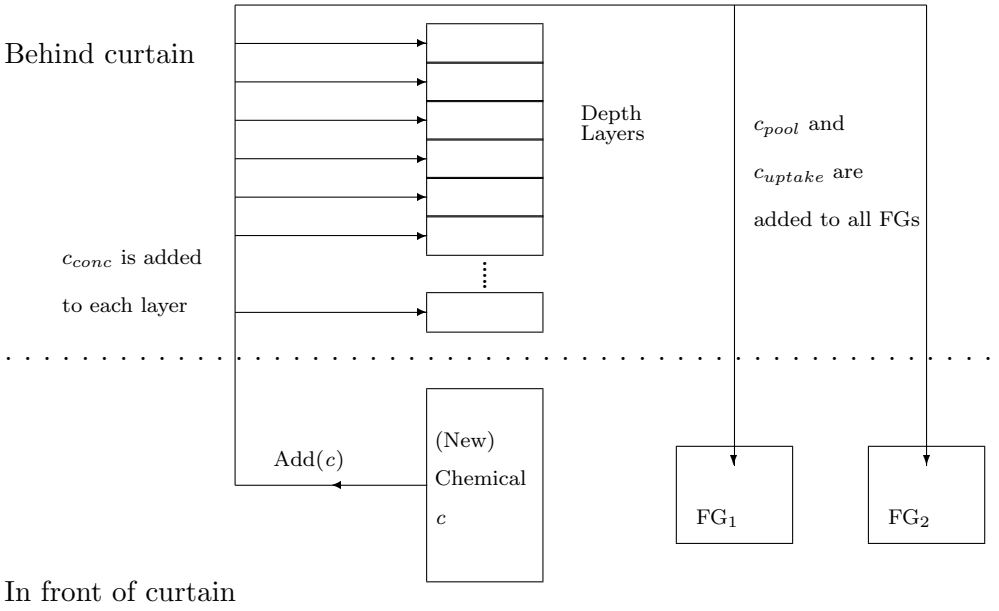

Figure 3.4: Introducing a chemical, c, to a model containing two functional groups

The effect of turbulence is approximated by averaging the concentrations of all chemicals from the surface to the turbocline; this is done automatically each timestep.

The ambient concentration of a chemical for a given particle is the concentration at the nearest grid point to the particle. Particles can transfer chemicals between their

internal pool and the environment, but to allow automatic conservation of chemicals, users cannot write to the chemical concentration directly.

When many particles uptake chemicals in the same layer, the total amount of chemical requested might exceed the amount of chemical available. In this case, depletion has occurred, and conservation of chemicals is enforced by the metamodel. Since particle-to-particle interaction is not permitted, there is no prior way of predicting whether depletion will occur. Figure 3.5 shows the method of correction for chemical concentrations that have been depleted.

$$uptake(c, x) \tag{3.1}$$
$$where \quad c = name\ of\ chemical$$
$$x = amount\ to\ uptake\ (micrograms)$$

The update function issues a request for chemical, which is stored internally. At the end of the timestep, when all requests have been made, the concentration is updated, and if it became negative, a depletion error occurred. All the particles that absorbed that chemical are revisited, and have their c_{uptake} variable adjusted, such that the total of all the uptakes for that chemical in that layer is equal to the initial amount of chemical available. The amount of chemical the particle absorbed is stored in the c_{uptake} variable within the particle, which the user can read. The ambient concentration is then set to zero.

Chemicals can also be released to the ambient environment. As c_{conc} cannot be written to directly, a special function is provided to increase the concentration of a chemical. It is similar to $uptake(c, -x)$, except that since depletion protection is not required when releasing chemicals, the registers described in figure 3.5 are not used when the release function is called.

$$release(c, x) \tag{3.2}$$
$$where \quad c = name\ of\ chemical$$
$$x = amount\ to\ release\ (micrograms)$$

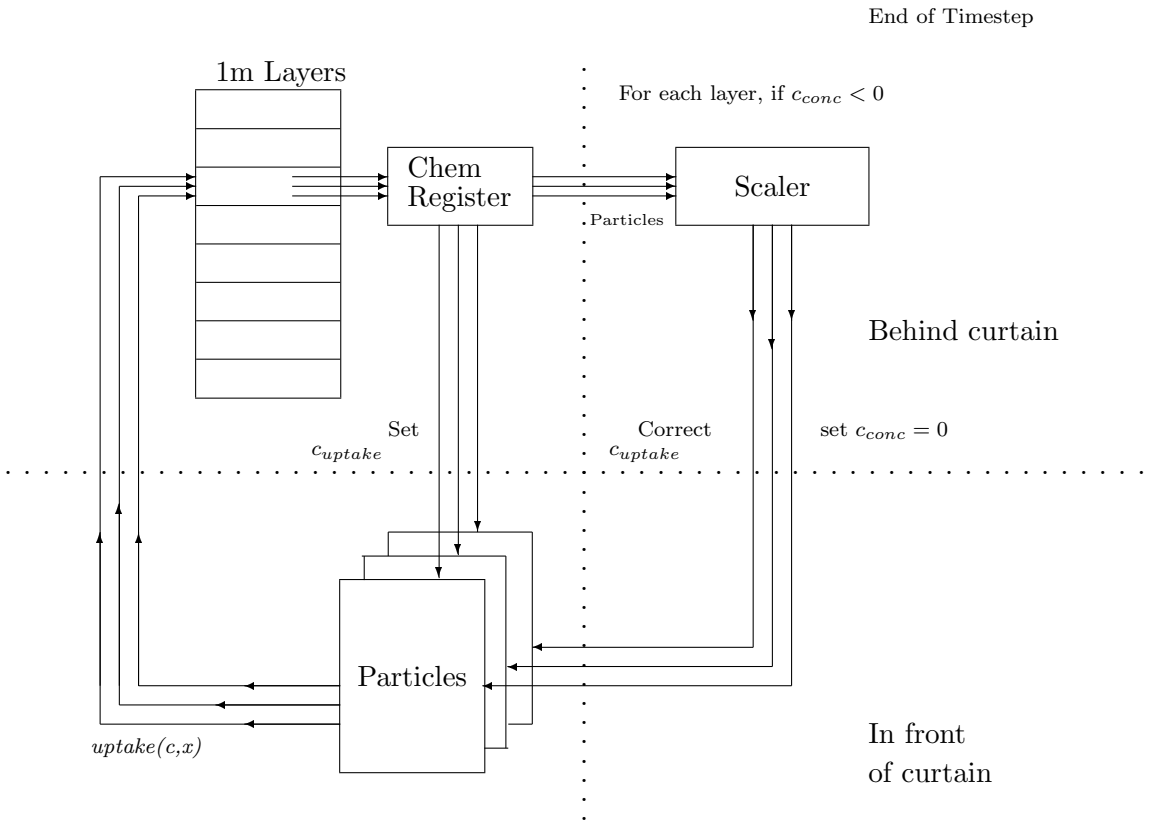

Figure 3.5: Chemical uptake, with behind the curtain depletion handling

Note that while the user models the chemistry of an individual, behind the curtain *uptake* and *release* are applied at the sub-population level. The quantity of each chemical to uptake or release is multiplied by the size of the sub-population, as all the individuals in the sub-population do the same thing.

3.3.3 Biofeedback

Some chemicals correspond to pigments and have associated action spectra. An action spectra defines a property of a chemical as a function of wavelength. The user can define the values for each of 25 wavebands, as described in section 6.3. The presence of pigments in solution, or within the pools of particles affects the way light is dissipated, thus

affecting the photosynthesis of the particles (self occlusion). This bio-optical feedback from the biology to the physics is managed automatically, and is mediated entirely through the pigments, using their action spectra.

3.4 Particles

A particle in the simulation is represented as a set of state variables. Some state variables are automatically created: these are the *pool* and *uptake* variables for each chemical, the depth, stage (see below), and type of the particle. Other state variables can be created by the user. In each timestep, a set of rules is executed for each particle, which define the new state of the particle in terms of its previous state, and its ambient environment.

3.4.1 Staged Growth

In nature, organisms grow through a number of stages in a life-cycle. For example, copepods are born in infancy, a proportion survive to become juveniles, then they grow to become adults, at which point they reproduce, and then they become senile, and eventually die. In each stage of their life-cycle, their behaviour changes. We model this behaviour by introducing the concept of a *stage*.

Particles in Planktonica may exist in different stages, and the rules describing one stage may be different to those in another stage. A functional group contains a set of rules, and the stage selects a subset of those rules that are to be executed in that stage.

Furthermore predators targeting a certain functional group may prefer to eat particles of one stage over another. Since particle-to-particle interaction is not permitted, the predator cannot establish the stage of a plankton. To provide such selective ingestion, the stage variable is handled behind the curtain, and predators can target a concentration of particles indexed by both type and stage.

Changes in Stage

A particle may also change its stage, either deterministically, or probabilistically, using the following two special functions.

$$change(s) \tag{3.3}$$
$$where \quad s = name\,of\,stage$$
$$pchange(p, s) \tag{3.4}$$
$$where \quad s = name\,of\,new\,stage$$
$$p = 'probability'\,of\,change$$

When the user writes a probabilistic stage change, they expect that the individual they are modelling has a fixed probability p of changing stage. Behind the curtain however, p represents a proportion of the sub-population that will change stage, hence the actual behaviour of the metamodel is that shown in figure 3.6; the sub-population splits into two, the proportion p of the individuals assuming the specified stage.

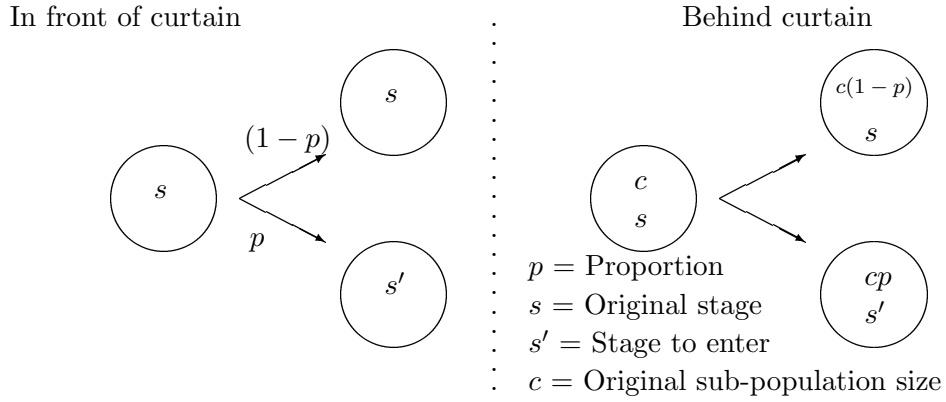

Figure 3.6: Probabilistic Stage Change

Implementation Detail

Note that the user could specify stages using a state variable representing the current stage of a particle, which could be used when defining the rules for that particle. How-

ever, this would not have supported selective ingestion of particles of a certain stage; as particle-to-particle interaction is forbidden, the stage of a potential prey is not visible to the predator.

Furthermore, making stages a metamodel property allows a minor performance improvements to be made; the particle update for each stage is represented in a separate Java class, removing the need to check whether each rule should be executed in the current stage.

Creating New Particles

Particles may spawn other particles of the same functional group, but the spawned particles may assume any one of the parent's stages. Additionally, the state variables of the spawned particles may on creation have different value to those of the parent.

$$create(s, n, [a]) \tag{3.5}$$
$$where \quad s = stage\ of\ spawned\ particle$$
$$n = number\ of\ individuals\ to\ create$$
$$[a] = assignments\ for\ offspring\ state$$

The *create* function is provided for this purpose. By default, the state variables of the spawned particles will be copies of those of the parent, but the list of assignments, $[a]$, allows the user to set any number of state variables for the offspring, to values based on the parent's state, and the ambient environment. Thus, in figure 3.7 below, the set $[a]$ maps the original state variables, v, to the new state variables v'.

While the user expects one individual to create n offspring, behind the curtain all the individuals of a sub-population of size c, produce n offspring each, and all the offspring together comprise a single new sub-population of size cn, which is independent from the parent.

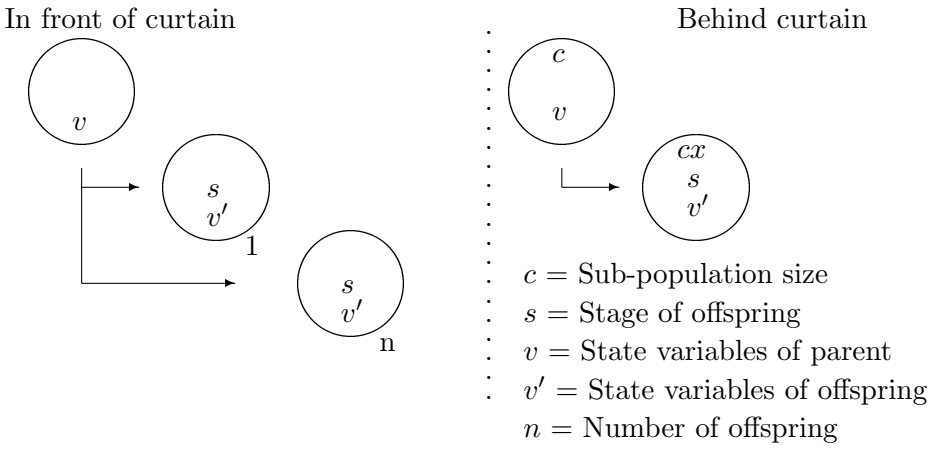

Figure 3.7: Creating new particles

3.4.2 Cell Division

Plant-based plankton grow by cell division rather than by reproduction. This is a different biological process from reproduction, and we model it separately.

$$divide(n) \tag{3.6}$$

$$where \quad n \; = \; number\ of\ cells\ to\ divide\ into$$

.

This specifies that an individual divides into n individuals, and the metamodel interprets this as all individuals of the sub-population dividing into n. Thus if the original cell count was c, the final sub-population size will be cn.

Note that a similar result could have been achieved using the *create* function, but the point here is to model biological processes. Furthermore, using the *create* function for every type of reproduction can be expensive; the prime example is during a phytoplankton bloom when the population of phytoplankton increases very sharply.

Particle Management

A fundamental difference between cell division and creation is that after creation, the parent and offspring are independent from each other; they have their own trajectories, meaning they have entirely separate state. Cell division by contrast results in more individuals, but they all have the same trajectory - the same state variables, moving together as one particle.

Apart from creation of new particles, the only way that the number of particles (agents) in the simulation changes is by particle management, which the metamodel supports. For each type (variety, described below) of plankton, and each stage, the user can specify how many independent sub-populations are to exist. Two methods are provided; the first specifies that if the number of particles exceeds a certain threshold, the smallest sub-populations should be merged until the number of particles is below that threshold. The second specifies that if the number of particles drops below a certain threshold, the largest sub-populations should be split until the population size is above that threshold. See section 7.4.

The user specifies these thresholds, which can be applied either to each layer in turn, or to the whole column at once.

3.4.3 Biodiversity

We define a functional group to be a class of organisms that for a given stage perform the same set of rules. A variety is a parameterisation of that functional group, and all individuals are defined as members of a variety. This models the biodiversity observed in nature, allowing the user to create many parameterisations of a functional group, rather than many separate functional groups.

Two levels of parameterisation are used, as demonstrated in figure 3.8. A functional group defines a rule, which in this case is a simple function of some parameter y. The species defines the relationship of y in the form ax^b, defining the values for a and b, for a given base parameter x, which is specified at the variety level. The form ax^b is chosen as it can exhibit a rich set of characteristics using only two parameters.

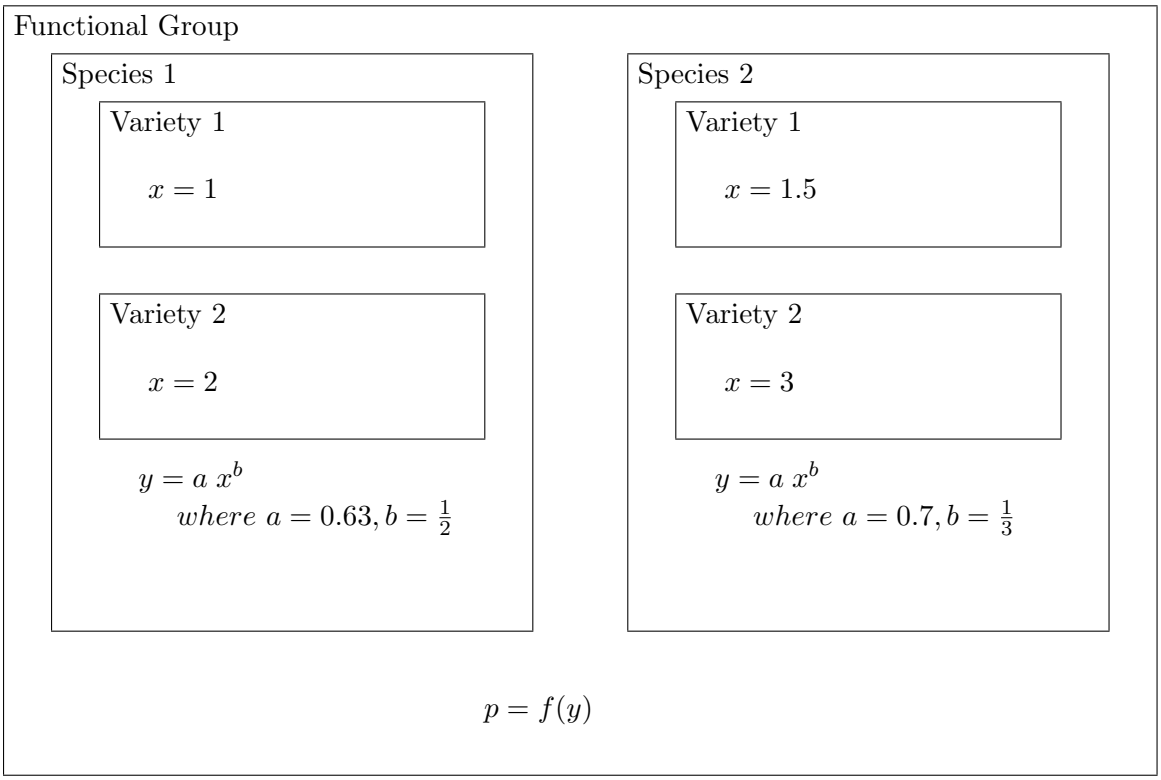

Figure 3.8: Functional Groups, Species and Varieties

3.4.4 Motion

The depth of a particle may be changed by model motion, e.g. by swimming, sinking or turbulent advection. All particle motion is presently handled by the user. In nature, particles may interact with their environment along the trajectory of motion. A special function is provided to sum a specified quantity between two depth values.

$$x = integrate(f) \tag{3.7}$$
$$where \quad x \ = \ finite\ sum\ of\ f\ over\ trajectory$$
$$f \ = \ any\ function$$

The function represents a finite sum over the distance the particle travelled in the previous timestep. Figure 3.9 shows how this is computed for function f. For example $integrate(1)$ returns the distance the particle travelled in the previous timestep using

the integrate function.

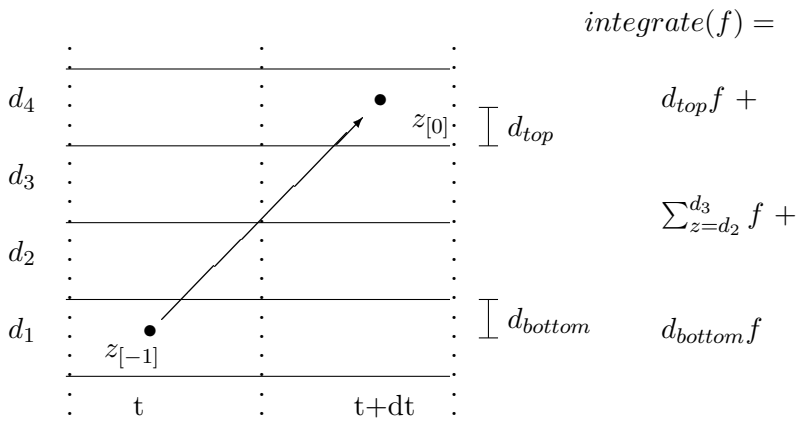

Figure 3.9: A finite sum for a particle's trajectory using integrate

Any variables in function f that depend on depth, are applied to the ambient environment of the particle, as it moves along its trajectory. As a contrived example, the following rule calculates the average temperature, (a depth-dependent property), of the water through which a particle travelled during a timestep.

$$T_{avg} = \frac{integrate(temperature)}{z - z_{[-1]}}$$

$$where \quad z = depth\ of\ particle\ at\ start\ of\ timestep\ t$$

$$z_{[-1]} = depth\ of\ particle\ at\ end\ of\ timestep\ (t-1)$$

Note also the notation of $z_{[-1]}$. The set of rules for a particle define its state at the start of timestep $(t+1)$, in terms of its state at the end of timestep t. When a variable like z is used on the right hand side of a rule, its value is that which z held at the *end* of the previous timestep. When $z_{[x]}$ is used (where x is negative), the value is taken from the end of timestep (t-x), and we say that z has a *history*.

The user can specify that any particle state variable should have a history, and it is the user's responsibility to define its size. This allows the modelling of particles with biological memory - see section 4.3 for further discussion about variable types.

3.4.5 Ingestion

A predator may feed on any individuals it encounters during a timestep. The result is that the sub-population sizes of the predated particles need to be reduced by an ingestion rate over the distance the predator travelled. This behaviour is provided with the *ingest* function:-

$$ingest(\overline{V}^*, \overline{r}) \tag{3.8}$$

Here, V^* can be thought of as a concentration of prey of a particular variety, in a particular stage, and r as the preferred rate of ingestion. Behind the curtain, the prey that the predator encountered as it moved along its trajectory each have their sub-population reduced according to the rate.

This is, however, a simplification. A model may consist of many different varieties in any number of stages. As defining an ingestion rule for each variety of prey in each stage would be cumbersome, we instead model sets of particles that are targets for ingestion. Using conventional mathematical notation, we therefore define $\overline{V}^*$ as a vector containing all the particle concentrations that the predator will eat, and $\overline{r}$ is defined as a vector of rates, one for each target in $\overline{V}^*$. The members of $\overline{V}^*$ and the associated values of r are defined in VEW Controller, a separate utility which allows the user to define the species and varieties for a model. This is described in detail in section 4.3.9.

3.4.6 Changing the Metamodel

Changes to the metamodel are expected to be rare, but may occur over time. Such changes will require re-programming in up to three places: the interface, the compiler, and the fixed simulation classes. They may also invalidate previous models unless particular care is taken to ensure backward compatibility.

However, more radical changes may be required. For example, the physical environment at present is fixed and provided as standard for all models. Other work [27, 44] is generalising this; for example, the physics code is being re-written in a rule-based way, allowing the user to choose either the standard physics environment as before, or instead

to write their own turbulence or optics equations. Moving from 1-D to 3-D space [44] may require some changes, since all particles will now be defined with three ordinates, as will their ambient environment. Note however that except for rules regarding motion, the biological rules will be equally applicable to a 1-D or a 3-D metamodel.

3.5 Summary

In this chapter, we have described the contents of the metamodel upon which all object models must be built. The metamodel incorporates the Lagrangian Ensemble method of integration which is individual-based in that rules are written for an individual plankter, but Ensemble-based, in that each agent in the simulation represents a sub-population of plankton with its own trajectory. The Lagrangian Ensemble method forbids particle-to-particle interaction.

The metamodel currently includes a water column, which is a one-dimensional grid. Each grid point contains physical properties for optics and turbulence, which are automatically updated. Functional Groups in an object model define the behaviour of a particle, whereas varieties define a parameterisation of a functional group. All particles exist at the variety level, and are also defined by a stage, which can represent a different categorisation of a variety, for example a different stage in a life cycle. Particles can selectively ingest on other populations, choosing the population to feed by variety and stage.

The chemicals in the environment are defined by the user. Each chemical is represented as a continuum, and the introduction of a chemical causes the automatic creation of an internal chemical pool for each particle, a concentration within each layer, and an *uptake* variable in each particle used for chemical conservation checking.

Where metamodel properties are required, namely sub-population size, write-access to chemical concentrations, particle creation, and handling the intermediate gridpoints a particle travels through, API functions are provided.

Chapter 4

Introduction to Modelling with Planktonica

This chapter describes the underlying principles of an object model; its components, variables, and the language used for writing rules.

A complete ocean model in the VEW is built by a range of integrated applications shown in figure 4.1. VEW Designer is the first, and is used to develop the particles and chemicals, and their accompanying variables and rules. The rules are composed of statements, mathematical functions, the special-purpose functions defined in the previous chapter, and a range of variable types.

VEW Designer encapsulates the modelling language, enforcing the semantics by limiting what the user can select or create.

4.1 Functional Groups and Functions

A functional group pulls together a set of functions that together define the behaviour of all plankton that are members of that functional group. Each functional group has functions that define what the functional group does, and an internal state defined by a set of variables. It may also have associated parameters.

A function consists of a set of *rules*. The end purpose of these rules is to update

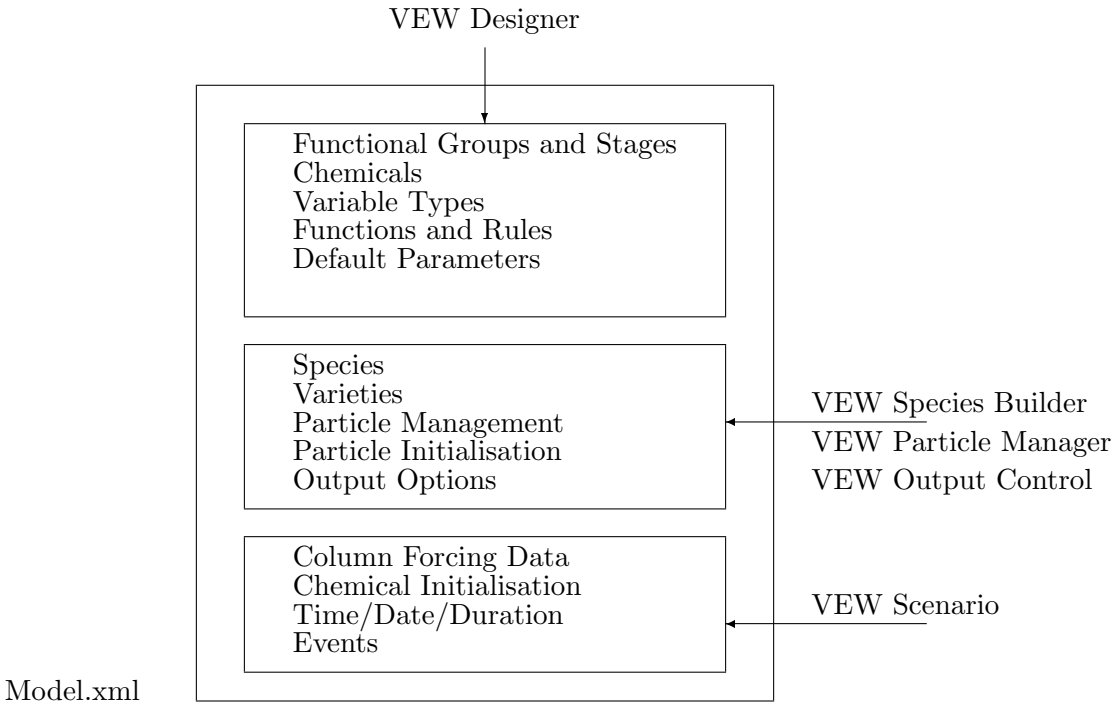

Figure 4.1: The Model Building Process

the state variables of a particle, using a combination of ambient physical and chemical properties, the functional group's own parameters, and previous values of its state variables.

Functions can also use local variables to split up large rules, or to re-use a result within that function. Sub-functions are like functions but can additionally export a result so that all functions can read that result as soon as it is calculated.

4.2 The State of the Simulation

The state of a simulation is defined by the following:-

- The physical properties of each layer in the water column (density, temperature, etc), and the turbocline depth.
- The concentrations of each chemical in each layer of the water column, and any

variables the user has associated with a chemical.

- The state variables of every particle in the water column, including its depth.

4.2.1 Ordering of rules

In each timestep, a set of rules is executed; the rules within the set are defined by the type of the particle being updated, and its current stage. Variables can only be written to once in each timestep, and for state variables, the effect of writing is delayed until the end of the timestep (*buffering*). This property is enforced by the user interface.

Planktonica aims to make the order of rules, functions and sub-functions irrelevant. However, the order in which *change* and *pchange* functions are called within the set of rules *is* significant when two statements that each cause a stage change. In this case, the stage assumed by a particle at the end of the timestep is the most recent of the stage changes (i.e., the last stage change in the list of rules).

In the case of probabilistic stage changes, the changes are treated in order, hence if the following two commands are performed in order on a particle that initially has a sub-population size of c:-

$$pchange(Stage1, p)$$
$$pchange(Stage2, q)$$

then the result is that cp individuals assume $Stage1$, and $c(1-p)q$ assume $Stage2$.

It may be argued that the ordering of *all* rules should be irrelevant, reducing the potential for error. If the two *pchange* rules above existed in different functions, then their order could easily be overlooked.

However, the solution would be to combine all the stage changes, probabilistic or not, into one rule that specified the stage in the next timestep. This would have an adverse effect on the modularity of the model, since the stage changes associated with different aspects of plankton behaviour, (for example, old age, predation, infant mortality and energy loss) would need to be combined into one rule. This would provide the simplicity

of order-free rules, but at the cost of removing model modularity when considering changes in stage.

The choice has been made to promote modularity and allow the user to write multiple stage changes, each in the function most relevant to the stage change. The cost is that the user must be aware that stage changes are executed from top to bottom through the sub-functions, then top to bottom through the functions.

4.2.2 Buffering

The buffering of variables ensures that the order in which the particles are updated does not give an artificial advantage to any plankton. For example, when plankton compete for a limited supply of nutrients, it is important that the nutrients are not absorbed in a 'first-come first-served' way, such that the first plankter updated has a higher likelihood of absorbing nutrient than those updated afterwards. The use of buffering allows certain generic corrections, such as depletion handling, to be carried out between timesteps as described in section 3.5.

Timesteps are distinct; the state of the simulation changes only at the boundary between timesteps. This provision aims to make future parallelisation of the simulation code straightforward.

4.3 Variable Types

An object model consists of functional groups. Each functional group contains functions and sub-functions, and each of those contains rules similar to mathematical equations. To construct these rules, a range of variable types are required. The different variable types and their scopes are shown in figure 4.2, and are described below. Note that the user never sees the physical or biological layer structures; variables are indexed only by numerical depth.

An individual in the simulation has a set of state variables, and a set of parameters. It has a set of rules that define its behaviour, which are segmented into functions and sub-

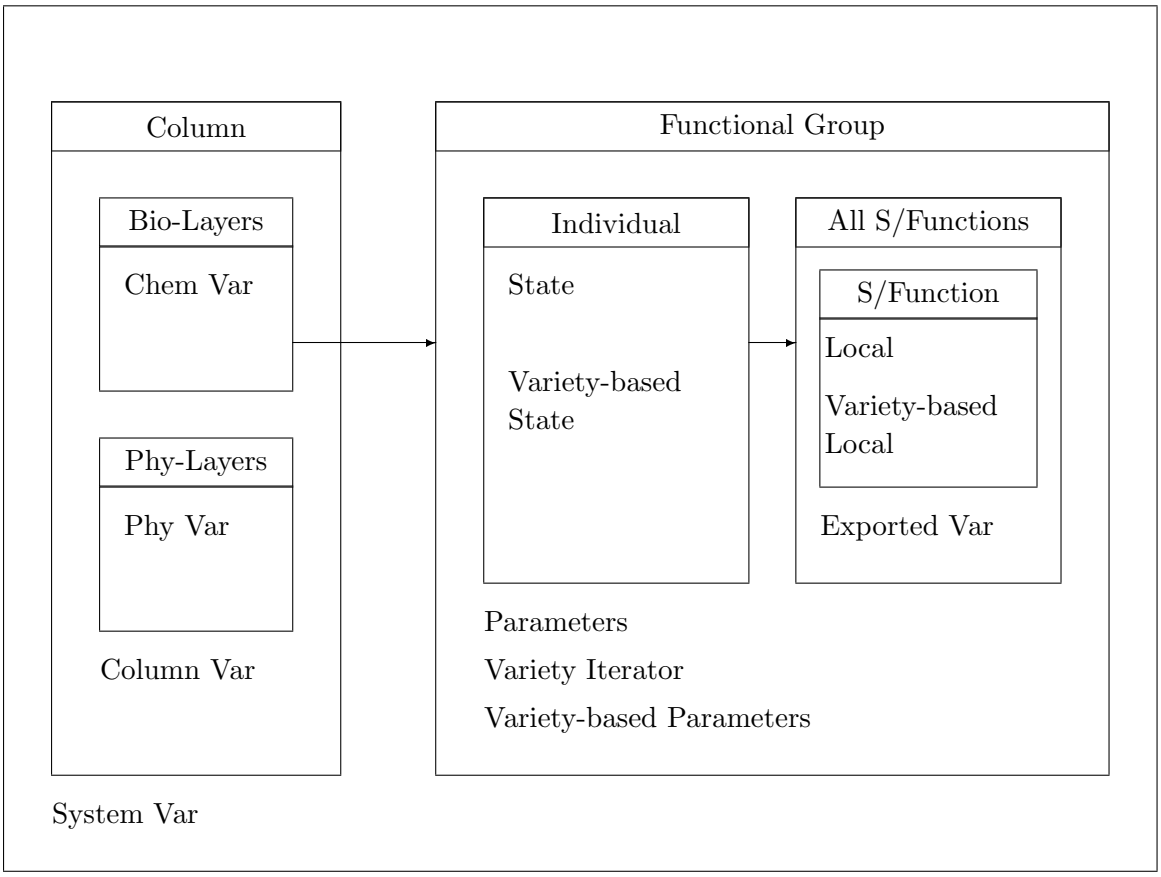

Figure 4.2: Variable types and their scope

functions. Local variables may store values between two rules, and exported variables can be assigned in a sub-function, and used within any function. Variety-based variables are used primarily for allowing a particle to target a set of populations, defined by variety and stage, for ingestion.

The variables types for biology are described in the following sections. By way of introduction, the following sequence of steps describe how a user should decide what variable type is required, when introducing a new variable, or using an existing one.

1. Is the user creating a new variable, or referring to a variable already provided within the simulation? If a new variable is to be created, move to step 6.

2. Visible irradiance, full irradiance, density, salinity, or temperature are the **physics**

variables in the particle's ambient (local) environment.

3. If the variable is a concentration of a chemical the user has added, then the variable is a **chemistry variable**.

4. The turbocline depth, *MLDepth* is a **column variable**.

5. The timestep size, $\triangle t$, and other built-in constants such as π, are **system variables**.

6. Does the variable apply purely to the individual being updated, or should there be a separate instance of the variable for each of a set of particle types? If the latter, the variable is **variety-based** - move to step 11.

7. Is the variable a constant property of the individual being updated? If so, it is a **parameter**.

8. Is the variable used to store a value for use in the same function, within the same timestep? If so, it is a **local variable**.

9. Is the variable to be used in more than one function, but within the same timestep? If so, use an **exported** variable, and create it within a sub-function.

10. Is the variable a property of the individual that changes each timestep? Is it necessary to recall values from previous timesteps? In either case, a **state variable** should be used.

11. The variable is variety-based: should it define a set of particle-types, for example, to be ingested? If so, then it is a **variety iterator**.

12. Should the variable contain a constant value for each member of a variety-iterator? If so, it is a **variety-based parameter**.

13. Is the variety-based variable used to store a set of values, one for each member of a variety-iterator, which is to be read only in the same function, within the same timestep? If so, it is a **variety-based local variable**.

14. Finally, if a state variable is required for each member of a variety-iterator, then create a **variety-based state variable**.

.

4.3.1 State Variables

State variables are created at the functional group level. They define the state of an individual. They are buffered, so when they appear on the left hand side of a rule, the effect of that assignment is delayed until the end of the timestep. When they appear on the right hand side of a rule, the variable's value is that at the beginning of that timestep. The reason they are buffered is so that the order of reads and writes on a variable is irrelevant.

State variables also have a history, meaning that it is possible to retrieve the value of an internal state variable from earlier timesteps. The user specifies the size of the history when creating the variable. Such history-based state variables can also be used to store values of elements that have no history, for example physical properties like temperature.

Hidden Generic State Variables

A few state variables are generic and exist for all plankton. Some of these are hidden, namely *stage*, *type*, and *count* which are the growth stage, variety type and sub-population size of the particle respectively. These cannot be read or written to, but *stage* and *count* are changed indirectly by use of the metamodel API functions described in chapter 3. The sub-population size is kept hidden to force the user to think in terms of individuals rather than a sub-population, which is a metamodel feature. The *stage* variable stores the growth stage in an internal format, which is better exposed through an interface menu than by making the variable visible. Similarly, the *type* variable is meta-data about a particle, used behind the curtain primarily for system calls such as *ingest*, and also for particle management. The user need not be aware of this variable when writing rules.

An additional hidden variable, $C_{release}$ is added for each chemical C that the user introduces. This is used purely for keeping track of chemicals released to the environment by the particle. It is hidden (i.e., cannot be referred to by the user) to ensure conservation of chemicals; the user cannot release chemicals to the environment without using the *release* special function, thus the kernel controls transfer of chemicals between particles and the environment. Since the $C_{release}$ variable is not changed behind the curtain, it is not necessary to expose it; this differs from the C_{uptake} variable described below, which might be changed by the metamodel's depletion prevention.

Read-Only Generic State Variables

The C_{uptake} variable, which exists for each chemical C, is visible to the user, but read-only; the user can only write to it indirectly using the *uptake* special function. This is because C_{uptake} has a special purpose for handling chemical depletion, and all changes to it must be logged behind the curtain. Reading the C_{uptake} variable gives the amount of chemical gained by uptake in the previous timestep, which may have been adjusted if chemical depletion occurred. See section 3.3.2.

Standard Generic State Variables

Finally, some generic variables can be read and written. These are the depth of the particle, z and the internal pool C_{pool} for chemical C. Note that C_{pool} is not automatically affected by the *release* or *uptake* functions. While it may seem intuitive to reduce the chemical pool automatically whenever chemical is transferred between the particle and its ambient environment, forcing this behaviour would prevent certain biological or chemical processes from being modelled. For example, some plants can increase the concentration of an internal chemical pool, perhaps using other chemicals they have absorbed. When modelling disease, for example, a chemical can by used to represent a virus that may be contracted and later reproduce within a plankter, thus the amount released will be more than the amount absorbed.

Therefore, the user has full control over the C_{pool} variables. When a plankter is

ingested by a predator, the predator automatically absorbs all the chemicals from the prey's pools into its own chemical pools.

4.3.2 Parameters (Constants)

A parameter is a named constant shared between all the members of a functional group. The VEW Controller allows many instances of each functional group to be created, and allows the parameter values in some instances to be different than in others - see the description of biodiversity in section 3.4.3.

4.3.3 Local Variables

The user can break down a complex equation or rule into a number of simpler steps, storing the values in local variables. Such local variables are provided for this reason, and exist only within the scope of a single function or sub-function.

4.3.4 System Variables

These variables can be considered as meta-data for the simulation, for example, the size of the timestep which is represented by ($\triangle t$).

4.3.5 Exported Variables

The sub-functions of a functional group are, by definition, executed before the functions of that group. The user defines a sub-function when the result of some rule is required in the same timestep. The exported variable is defined within the sub-function for this purpose: it stores a value which can then be used in the functions executed immediately afterwards.

For example, suppose X is a plankton state variable which is affected by three other properties, A, B and C. For each of the three properties, suppose we have two methods of calculating that property, a_1, a_2, b_1 and so on. For example, it may be the energy change for a plankton is defined by three processes, photosynthesis, respiration,

and reproduction, and we have found in the literature two methods of calculating each process. Rule 4.1 shows a possible definition for X, without using subfunctions.

$$X = a_1 + b_2 + c_1 \tag{4.1}$$

If the method of photosynthesis is to be changed from a_1 to a_2, we would need to change the rule for X. Sub-functions and exported variables make it possible to express this rule in the form of rule 4.2.

$$X = A + B + C \tag{4.2}$$

$$\textit{where} \quad A = a_1 \tag{4.3}$$

$$B = b_2 \tag{4.4}$$

$$C = c_1 \tag{4.5}$$

Here, A, B and C are defined as exported variables, each one defined in a sub-function, but X is defined in a standard function. This causes A, B and C to be calculated before any standard functions are calculated. Hence, if X is defined within a standard function and A, B and C within sub-functions, then X can use the exported variables A, B and C immediately.

One benefit of this approach is that if we want to change the method of photosynthesis from a_1 to a_2, we only need to change the definition of A - a simpler task than updating X in the earlier example. Secondly the values A, B and C can be re-used in any number of functions.

4.3.6 Physics Variables

Referring back to figure 3.3, the built-in physics code exposes a number of ambient environment variables. These are read-only variables, and biofeedback is the only mechanism by which plankton may affect their ambient physical environment.

Physical variables can be logged, but do not have histories. However, their values can be recorded by assigning them to state variables, which can have histories. This has a number of uses, for example when modelling particles that have a memory of how the irradiance has varied over time.

4.3.7 Chemistry Variables

Recall from figure 3.4 that when a chemical is added, a concentration variable is added automatically to each layer in the water column. Such chemistry variables are read-only for reasons discussed; they can only be altered by use of the *uptake* and *release* functions.

4.3.8 Column Variables

Column variables are properties of the water column as a whole, including the turbocline (also called the mixing layer depth) and the forcing environmental data such as sunlight and wind speed. As new values are read each timestep, writing to them would have no effect, hence they are defined as read-only.

4.3.9 Variety-based Types

The Lagrangian Ensemble method forbids particle-to-particle interaction. This presents a challenge when considering ingestion, since particles need to be aware of the concentrations and types of the particles they are feeding on. The challenge is to allow the user to specify that an individual may aim to ingest different varieties, or different stages of plankton, at different rates. Recall also that rules are written at the functional group level, whereas the targets for ingestion are defined as varieties (see figure 3.2), and at the time of writing the ingestion rules, the varieties will not have been defined.

The solution is the provision of *variety-based types*, which are vectors, where each element is associated with a population-type indexed by variety and stage. We firstly define an *iteration vector* to be a list of such particle-types that an individual may ingest. A global structure exists behind the curtain which stores the concentrations of each particle-type, also indexed by variety and stage. The user can create any number of iteration vectors in Planktonica, and after the varieties have been defined, the user will select the elements for the iteration vector. Figure 4.3 shows an example configuration of an iteration vector. By convention all vector variables are overlined.

Recall that the first argument of the *ingest* function specifies the prey to be ingested (see section 3.4.5), defined by its variety and stage. The second argument of the *ingest*

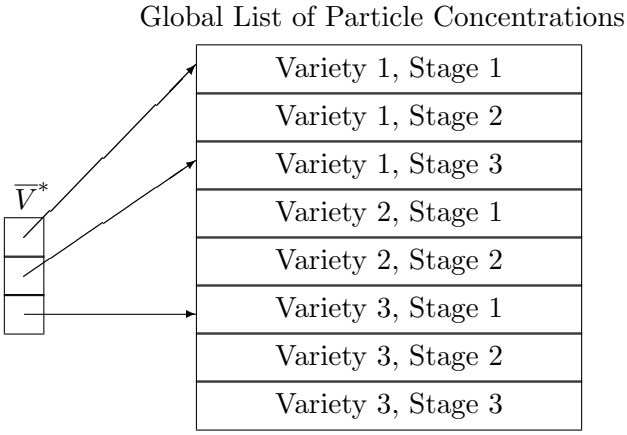

Figure 4.3: An example of an iterator vector and the global structure

function is an expression that may compute a different answer for each type of prey in the iterator. An *ingest* function can thus be considered an iteration for each element, i, of an iteration vector, where the second argument is a function of i, which defines the rate of ingestion for population-type i.

In order for the second argument of the *ingest* function to return a different answer for each type of prey in the iterator, there must be some identifiers in the second argument that will be indexed by i in the iteration. Three such variety-based types are provided, each of which the user must explicitly associate with a variety iterator when they are created. The first is a variety-based parameter: a constant defining for instance a separate rate of ingestion for each element in the iterator. The second is a variety-based state variable, allowing a state variable associated with each member of the iterator to be maintained. Finally, for convenience, variety-based local variables can be created, which store an intermediate value associated with each element in the iterator.

Figure 4.4 shows another example configuration; this time the user has defined two variety iterators of different length, and for each iterator, two more vectors, either vector-state variables, vector-parameters, or local vectors have been created. When creating vectors $\overline{A}$ and $\overline{B}$, the user has to explicitly state which iterator they are associated with, in this case $\overline{V}^*$.

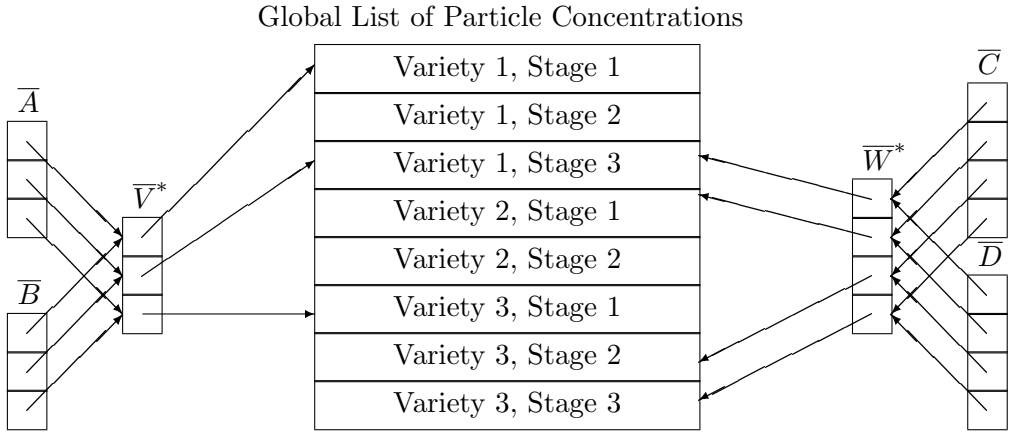

Figure 4.4: An example of an iterator vector and the global structure

In section 4.5.2, we will describe how such vectors behave when they occur in rules.

4.3.10 Histories

When creating state variables and variety-based state variables, the user may specify that such variables have a history, and the user can retrieve a value that the variable held a number of timesteps ago. The size of this history is specified by the user, and the value of a variable from a number of timesteps ago is retrieved with the expression:

$$varhist(v, e) \tag{4.6}$$

Here, v is the variable in question, and e is an expression for the difference between the current timestep, and the timestep from which to retrieve the value of v. A value of -1 for e returns the value a single timestep ago, -2 for two timesteps ago, and so on. Note that the result of e is a real number, and the value used for looking up the history is the nearest integer $i \leq e$.

As the result of e cannot be predicted, it is the user's responsibility to ensure that all history lookups are within the range they specified.

4.3.11 Naming Conventions

The names of variables are under the control of the user. They must begin with a letter of the alphabet, which may be followed by any number of alpha-numeric characters, or the underscore character. This is similar to conventional programming languages such as Java. Within planktonica the first underscore of any variable name causes the remainder of the variable name to be displayed as subscript.

Vectors are always labelled with an overline, e.g. $(\overline{V})$, and an iterator is labelled additionally with an asterisk, e.g. $(\overline{V}^*)$.

Planktonica forbids the creation of any duplicate variables in a model file, or variables with the same names as functional groups or chemicals.

4.3.12 Units

Variables and constants have a default value, and units. Concentrations are often written in either $mMolm^{-3}$ or μgm^{-3} and mistakes regarding these units are common. Planktonica requires units to be specified for every variable, and also for numerical values included in rules. Representing units, and checking them is not especially difficult and has been well studied elsewhere, for example [32]; when building rules, Planktonica provides a unit checking facility, which informs the user if any inconsistency in units is found.

4.3.13 Stages

New stages can be created at the user-interface level, and Planktonica ensures they are named uniquely to each other and to other variables and chemicals in the model. The user specifies that a function or sub-function is applicable in a set of stages by ticking the checkboxes for the appropriate stages. Individual rules cannot be set to run in different stages; if this is required, the rules should be separated into different functions or sub-functions.

Note that the implementation of stages uses a separate update class that overrides

a generic update class; this removes the need to perform a comparison for each rule to determine whether the rule is applicable in the current stage.

4.4 Differential Equations or Rules

Biological behaviour is usually specified by defining the internal state of an organism over time. As time is continuous, this typically amounts to defining the rate of change of each state variable via an ordinary or partial differential equation. This is generally what is seen in the literature.

However when integrating a model, time must be discretised. An alternative way of defining the biology is to state explicitly the value of each variable in the next timestep, as a function of its value in the current step. Both mechanisms are provided by Planktonica, but in some instances we have found the use of rules to be preferable. Consider the following example for turbulent motion, equation 4.7, and rule 4.8.

$$\frac{dz}{dt} = \frac{rnd(MLDepth) - z}{\triangle t} \tag{4.7}$$

$$dz = rnd(MLDepth) - z \tag{4.8}$$

These both have the same effect. However equation 4.7 is misleading as it implies that $\frac{dz}{dt}$ is a rate of change over time, which for turbulence is not the case; it is actually the change in distance after time $\triangle t$. Additionally, the need to divide by the timestep is far from obvious, requiring the user to predict that when integrating the rule, the right-hand side will be multiplied by the timestep.

4.5 Building Rules

This section describes the constructs available for building rules in front of the curtain. These are essentially mathematical constructs, and where behind-the-curtain access is required, the seven special functions described in chapter 3 are used. A rule is defined as one of the seven metamodel functions, an assignment, or a conditional execution, as described below.

4.5.1 Scalar Assignments

The user can create a scalar assignment of the form

$$\text{v=e} \tag{4.9}$$

Here, v can only be a state variable, or a local variable. Recall that since state variables are buffered, v will take the value of the numerical expression e at the beginning of the following timestep, whereas a local variable will assume the value of e immediately. The expression e must be scalar; it must return a single result. See section 4.5.5 regarding numerical expressions.

4.5.2 Variety-based (Vector) assignments

$$\overline{v} = e \tag{4.10}$$

If a vector appears on the left-hand side of a rule, then an iteration occurs, whereby a separate assignment is performed for each element of the vector. For a given element index, the expression e is evaluated with respect to the same index, such that all vectors referred to in e will be indexed identically. Referring back to figure 4.4, consider the following rule.

$$\overline{A} = \left(\overline{B} * \overline{V}^*\right) + 1 \tag{4.11}$$

Importantly, Planktonica enforces that whenever vector terms exist in a rule, they must all be linked to the same iterator, hence inclusion of $\overline{C}$, $\overline{D}$ or $\overline{W}^*$ would not be permitted in this rule. Since the vectors here all have an arity of 3, this assignment has the following meaning:-

$$\overline{A}_1 = \left(\overline{B}_1 * \overline{V}_1^*\right) + 1 \tag{4.12}$$

$$\overline{A}_2 = \left(\overline{B}_2 * \overline{V}_2^*\right) + 1 \tag{4.13}$$

$$\overline{A}_3 = \left(\overline{B}_3 * \overline{V}_3^*\right) + 1 \tag{4.14}$$

Note that the scalar quantity '1' is replicated in each iteration. This is permitted, as is $\overline{A} = 1$, which would cause all elements of $\overline{A}$ to take the value of '1'.

The left-hand side of a vector assignment can only be a variety-based variable or a variety-based local variable; variety-based parameters cannot be written to, and iterators have the special property of both defining which particle-types are to be considered, and when read they return the concentration of each particle type.

4.5.3 Differential Assignments

Both scalar and vector assignments can be expressed differentially, as mentioned in section 4.4. These assignments take the form:-

$$\frac{dv}{dt} = e \tag{4.15}$$

At present, Eulerian integration is carried out, meaning the value of v will change by $e \triangle t$ at the beginning of the next timestep. As the assignment defines a change, v cannot be a local variable, or a variety-based local variable; v must either be a state variable or a variety-based state variable.

4.5.4 Conditional Execution

A conditional statement takes the form:

$$if\ b\ then\ F_1\ else\ F_2 \tag{4.16}$$

Boolean variables do not exist as first-class objects in Planktonica, but boolean expressions can be used in conditional equations, and here b is a boolean expression. Note that statements cannot be a part of an iteration, hence b must return a single result resulting in either statement F_1 or F_2 being executed once. See section 4.5.6.

Functions F_1 and F_2 can be either one of the seven metamodel functions, (change, create, divide, ingest, pchange, release and uptake), or an assignment (scalar or vector), a differential assignment, or another conditional.

4.5.5 Numerical Expressions

A range of standard mathematical expressions (abs, acos, asin, atan, cos, exp, log, log10, minus, pow, sin, sqrt and tan) are provided, and behave in the standard way.

The choice of standard expressions is motivated by Java's standard maths class. In planktonica's interface, addition, subtraction, multiplication and division are specified in a prefix notation, using the functions add, sub, mul and div.

Random numbers are provided using the *rnd* function; the implementation used is the Marsenne Twister [19], which offers better performance than Java's standard random number generator, yet offering a very large period of $(2^{19937}) - 1$.

An alternative conditional is also provided, which can occur within another expression (i.e., inline), and rather than causing a function to execute, it returns one of two values depending on whether the condition succeeds. It takes the form:-

$$if\ b\ then\ e_1\ else\ e_2 \tag{4.17}$$

Differently to the conditional statements, this conditional function can contain vectors, in both the boolean expression b, and the two expressions. Such terms will be indexed as demonstrated in section 4.5.2.

The *varhist* function mentioned provides access to the previous values of variables that have associated histories. Absolute scalar values can be used in numerical expressions, and any of the variable types described in this chapter can be read. The metamodel *integrate* function described in section 3.4.4 is also available.

Finally, three special purpose numerical expressions reduce vectors to scalars; these are $vAvg$, $vMul$ and $vSum$, which take the whole of a vector, average, multiply, or sum its elements, and return a scalar result.

4.5.6 Boolean Expressions

Boolean expressions are used in the conditional execution, (section 4.5.4) and inline as part of a conditional numerical expression. Standard comparison operators are provided $(=, \neq, <, >, \leq, \geq)$, which each take numerical expressions as their arguments in the expected way. Standard logical operators (*and*, *or* and *not*) are provided that take boolean expressions as their arguments.

Recall that for the conditional execution statement, only one boolean result is permitted, hence none of the numerical expressions must be vectors in such a conditional.

However, for the conditional numerical expression, (i.e., within an assignment), vectors are permitted, and the iteration occurs as in section 4.5.2.

Three special boolean operators are provided which reduce vectors to scalars. These are *none*, *all* and *some*. Given a boolean expression, in which the numerical expressions may contain vectors, these operators return true if none, every, or at least one of the iterations through the vectors gave a positive result respectively.

4.6 Chemistry Rules

Chemicals can also have update rules, which are a subset of the rules available for particles. The important difference is that while particle rules result in the update of an individual, chemistry rules result in the update of a chemical concentration, and other user-defined chemical variables within a biological layer. As a result, the following functions and variable types that only apply to particles are not appropriate for chemistry rules:-

- The seven special function calls (*change*, *create*, *divide*, *ingest*, *pchange*, *release* and *uptake*).
- The integrate function, since this involves particle trajectories.
- The variety-based types (variety iterator and variety-based parameters, local variables, and state variables)
- The variety-based numerical functions, $vSum$, $vMul$ and $vAvg$.
- The variety-based boolean functions, *all*, *none* and *some*.

Recall from section 3.3.2 that particle updates cannot write directly to chemistry concentrations; they can only indirectly affect the chemistry concentrations via *uptake* and *release*. Chemistry rules differ from this: they can directly write to the chemical concentrations. Moreover the c_{conc} variables are the only variables visible both to

chemistry and biology rules, and the end purpose of chemical rules is to update this variable.

The subset of the modelling language that is supported by chemistry rules, is as follows:-

- Functions and sub-functions can be created.
- Assignments, differential assignments, and conditional assignments are available.
- Chemical state variables are supported, allowing new variables, with a history, to be introduced to each biological layer.
- Chemical local variables and chemical exported variables can be created, and behave equivalently to those of biological rules.
- Parameters for chemistry rules can be created.
- All mathematical functions that do not apply to trajectories or variety-based variables are available.

Chapter 5

Semantics of Modelling Language

5.1 Introduction

This chapter defines formally the syntax and semantics of the modelling language. The operational semantics is given in the form of rewrite rules that collectively define how the state of a simulation is modified by the execution of rules defining the biology and chemistry of the water column. The notation is modelled on that of the functional language Haskell [17].

The language for defining chemistry is a subset of that for biology. This is because the biology is modelled with particles whilst the chemistry is modelled by fields. For this reason, we focus on the biological modelling language; once this is understood, the operational semantics of the chemistry modelling language follows straightforwardly.

Recall that biological organisms are instances of some variety and each may have a number of growth stages. For each variety and each stage there is a sequence of rules that constitutes the "update" code for all particles that are instances of that variety and which exist in that stage. The update code is simply a set of rules that will be executed in sequence by the Planktonica kernel for each such particle. The effect of each rule is to modify the internal state of the particle. The main objective of this chapter is to explain formally how this is done.

5.2 Syntax

The biology modelling language includes the primitive functions outlined in Chapter 3 for supporting the Lagrangian Ensemble metamodel, and support for variety iterators and variety-based variables (vectors) linked to them. Figure 5.2 summarises the syntax of the biological modelling language. Id denotes the set of variable identifiers, including overlined variety-based identifiers, and subscripted identifiers. Num denotes the set of double-precision floating-point constants. Observe that multiplication of two expressions is represented by their juxtaposition: $x\,y$ is interpreted as $x \times y$, for example. The syntax of an "update" script is defined by *Rules* in Figure 5.2.

5.3 Model State

Model integration proceeds in timesteps. The operational semantics detailed below defines how the state of the model is updated during a timestep by the execution of one or more rules.

Chemicals and particle concentrations exist as continuum fields and these are modelled as concentrations at each point in a one-dimensional chemistry grid. A separate grid, with different internal structure, is used to model additional fields for the physics. The physics is built in and cannot be changed by the user. The two grids have different internal structure; importantly, the chemistry grid is regular, with L grid points and grid spacing h (metres).

The chemistry grid implicitly defines "layers" in the water column, with the grid points sitting at the centres of the various layers. When thinking of particles we will often refer to its ambient (chemical) environment as being its associated "layer" in the chemistry grid.

The dynamic state of a model is defined by the following:-

- The state of the physics grid, including the turbocline
- The state of the chemistry grid

$$
\begin{array}{lll}
Rules & ::= & Rule\ '{;}'\ |\ Rule\ '{;}'\ Rules \\
Rule & ::= & Assign| \\
 & & \frac{'d'\ Id}{'dt'}\ '{=}'\ Expr\ | \\
 & & 'if'\ BExpr\ 'then'\ Rule\ 'else'\ Rule\ | \\
 & & 'uptake'\ '('\ Id\ '{,}'\ Expr\ ')'\ i| \\
 & & 'release'\ '('\ Id\ '{,}'\ Expr\ ')'\ | \\
 & & 'ingest'\ '('\ Id\ '{,}'\ Expr\ ')'\ | \\
 & & 'change'\ '('\ Id\ ')'\ | \\
 & & 'pchange'\ '('\ Id\ '{,}'\ Expr\ ')'\ | \\
 & & 'divide'\ '('\ Expr\ ')'\ | \\
 & & Create \\
Assign & ::= & Id\ '{=}'\ Expr \\
AssignList & ::= & Assign\ |\ Assign\ '{,}'\ AssignList \\
Create & ::= & 'create'\ '('\ Expr\ '{,}'\ Id\ ')'\ | \\
 & & 'create'\ '('\ Expr\ '{,}'\ Id\ ')'\ 'with'\ AssignList \\
Expr & ::= & Num\ | \\
 & & Id\ | \\
 & & Id_{'['Expr']'}\ | \\
 & & 'if'\ BExpr\ 'then'\ Expr\ 'else'\ Expr\ | \\
 & & Expr\ Op\ Expr\ | \\
 & & Expr\ Expr\ | \\
 & & \frac{Expr}{Expr}\ | \\
 & & Expr^{Expr}\ | \\
 & & VOp\ '('\ Expr\ ')'\ | \\
 & & Prim\ '('\ Expr\ ')'\ | \\
 & & Prim2\ '('\ Expr\ '{,}'\ Expr\ ')'\ | \\
 & & 'integrate'\ '('\ Expr\ ')'
\end{array}
$$

Figure 5.1: Particle modelling language syntax (part 1)

$$
\begin{array}{lll}
BExpr & ::= & Expr\ \ Comp\ \ Expr\ \ | \\
 & & 'not'\ \ '('\ \ BExpr\ \ ')'\ \ | \\
 & & BExpr\ \ BOp\ \ BExpr\ \ | \\
 & & VBOp\ \ '('\ \ BExpr\ \ ')'\ \ | \\
Prim & ::= & 'abs'\ \ |\ \ 'acos'\ \ |\ \ 'asin'\ \ |\ \ 'atan'\ \ |\ \ 'cos'\ \ | \\
 & & 'exp'\ \ |\ \ 'log'\ \ |\ \ 'log10'\ \ |\ \ 'rnd'\ \ |\ \ 'sin'\ \ | \\
 & & 'sqrt'\ \ |\ \ 'tan' \\
Prim2 & ::= & 'max'\ \ |\ \ \ 'min' \\
Op & ::= & '+'\ \ |\ \ \ '-' \\
Comp & ::= & '='\ \ |\ '\neq'\ \ |\ '>'\ \ |\ '<'\ \ |\ '\geq'\ \ |\ '\leq' \\
BOp & ::= & 'and'\ \ |\ 'or' \\
VOp & ::= & 'vAvg'\ \ |\ 'vMul'\ \ |\ 'vSum' \\
VBOp & ::= & 'all'\ \ |\ 'some'\ \ |\ 'none'
\end{array}
$$

Figure 5.2: Particle modelling language syntax (part 2)

- The state of each particle in the simulation

.

Note that there may be additional static information (e.g. the rules for different varieties and stages, a pigment's action spectra etc.) but as none of this can be modified during execution we exclude it from the discussion.

5.3.1 Physics

The set of physics variables at each point on the physics grid is given by:

$$\begin{aligned} PhVars = \quad \{ \quad & Temp, \\ & Density, \\ & Salinity, \\ & FullIrrad, \\ & VisIrrad \quad \} \end{aligned}$$

The values assumed by these variables at each grid point are maintained in a *physics environment variable* which is a function of type $PEnv = Depth \rightarrow PhVars \rightarrow Double$, that delivers the value of variable $p \in PhVars$ at a specified depth. Note that in one dimension $Depth$ is synonymous with $Double$. Note also that we do not work explicitly with grid points in the physics.

For example, $\rho_{phys}\ 0.7\ Temp$ delivers the temperature (in $°C$) at depth $0.7m$ in the physics environment (grid) defined by ρ_{phys}.

5.3.2 Chemistry

The values of the chemistry variables for each chemistry grid point (layer) are maintained in a *chemistry environment*. Recall that the user can define new chemistry by the introduction of named chemicals and associated state variables, local variables and parameters. Recall also that pigments are simply modelled as chemicals with action

spectra. We therefore do not need to talk about pigments explicitly. Define:

$Chem$	The set of chemical names
$Pig \subseteq Chem$	The set of pigment names
$CState$	The set of chemical state variable identifiers
$CLocal$	The set of chemical local variable identifiers
$CConst$	The set of chemical parameter names (constants)

Recall also that state variables (here members of $CState$) may have associated histories, meaning that their value a specified number of timesteps earlier can be retrieved. For a variable v with an associated history, a reference to v itself in the language refers to the value in the current timestep; $v_{[-1]}$ represents the value in the previous timestep, and so on.

The history associated with a variable v is stored internally in a vector which we shall consistently label v'. The elements of these internal vectors can be accessed via a vector indexing operator ($\downarrow$). By convention, history vectors are indexed internally from -1 (downwards), in keeping with the source language. A special element $(v \downarrow 0)$, which cannot be accessed directly by the user, is used to store the value that the variable will assume in the following timestep. We overload the meaning of $\downarrow$ so that $v \downarrow i := e$ has the effect of assigning the i^{th} element of v to the value of e.

Different state variables may have different history lengths so we define H_v to be the history *index set* for v; this includes the special index 0. For example, a variable v with a three-timestep history will have a history index set $H_v = \{0, -1, -2, -3\}$. Thus,

$$CState' = \cup_{v \in CState} \{ v, v' \} \cup \{ c_{conc} \mid c \in Chem \}$$

which extends $CState$ to include the history vectors. Note that the index set (H_v) associated with v is specified by the user when v is declared in the Planktonica interface.

Introducing a new chemical $c \in Chem$ automatically introduces a chemical concentration variable C_{conc} at each chemistry grid point. Thus, the set of variables defined in a chemistry environment (grid) is given by

$$CVars = CState' \cup CLocal \cup CConst$$

A chemistry environment has type $CEnv = [CEnv']$ where $CEnv' = CVars \rightarrow Double$ and where $[a]$ denotes the type "list of a". $CEnv$ thus comprises a list of value mapping functions, one for each chemistry grid point. For convenience, if ρ_{chem} is a chemistry environment, we will write $\rho_{chem,i}$ to denote the i^{th} element of the list ρ_{chem}, which is the value mapping function for grid point $i, 0 \leq i \leq L-1$. It is also convenient to be able to index the chemistry environment by depth, so we define $\rho_{chem}@z$ to mean $\rho_{chem,\lfloor z/h \rfloor}$ where h is the chemistry grid spacing.

For example, if $Nitrate \in Chem$ then $\rho_{chem,12}$ $(Nitrate_{conc})$ denotes the concentration of $Nitrate$ in layer 12 of the chemistry grid whose state is given by ρ_{chem}. In general, parentheses around arguments in function applications may be optionally omitted (currying) where the meaning is unaffected.

Note that differently to the rules for particles which can only indirectly change chemical concentrations via built-in functions, chemistry rules can directly write to c_{conc} variables.

5.3.3 Biology (Particles)

Each particle represents a sub-population of some variety. Varieties are also instances of functional groups, but this is unimportant from the point of view of the semantics as the rules for all varieties of the same functional group are the same. Such varieties differ only in their parameterisation.

As with chemistry, new varieties (functional groups) can be introduced by the user, each having an associated set of state variables, local variables and parameters. Let N_v

denote the number of varieties and define for each variety $1 \leq i \leq N_v$:

$State_i$	The set of state variable identifiers
$Local_i$	The set of local variable identifiers
$Const_i$	The set of parameter names (constants)
$VIter_i$	The set of variety iterator identifiers
$VState_i$	The set of variety-based state variable identifiers
$VLocal_i$	The set of variety-based local variable identifiers
$VConst_i$	The set of variety-based parameter names (constants)

These sets contain all the variables defined by the user. Behind the scenes additional variables are implicitly defined in terms of these, as we now define.

Recall in section 4.3.5, *exported variables* were described. For the semantics, these can be considered equivalent to local variables, since the segmentation of rules into functions and subfunctions is purely for model-building convenience; in the semantics we ignore the segmentation and deal with a single list of rules, thus making export variables equivalent to local variables.

Internally, each variety is uniquely numbered so, for convenience, we define the index set $Var = \{ 1, 2, ..., N_v \}$ to index the various varieties.

Variety-based variables (prefix 'V' above) are stored internally as vectors of (scalar) double-precision floating-point numbers. These will be referred to as "variety vectors". The vector indexing operator ($\downarrow$) previously used for indexing histories will also be used to index variety vectors. However, variety vectors are indexed from 1. Members of $VState$ additionally have a history for each element of its internal variety vector – in other words, it is represented internally by a two-dimensional structure (a vector of vectors).

For variety $v \in Var$ let S_v be the number of growth stages associated with v. Internally, the stages are numbered $1, ..., S_v$, although the user refers to them by name. The names for the various stages are specified in the Planktonica user interface when they are defined. We will assume a function $S : Var \rightarrow Id \rightarrow Int$ for mapping these stage names to numbers.

Each variety automatically has associated with it default variables denoting its variety, stage number, cell count and depth (with history), together with pool, uptake and release variables for each chemical in the set $Chem$. The latter may also have associated histories. The *stage* and *count* variables both have "current" timestep and "next timestep" variants as they may be updated indirectly via the special metamodel functions. As the *stage* and *count* variables have no histories, we include the special identifiers $stage_{new}$ and $count_{new}$ to store the values that *stage* and *count* will assume in the following timestep.

All particles have a depth variable, and a depth history. The variety, stage and count variables are not directly accessible by the user – they can only be modified via the metamodel functions defined in Chapter 3. The uptake and release variables can be read by the user but only updated via the metamodel functions. The depth variable can be read and written directly by the user. The variable z denotes the current depth and z' its history vector with associated index set H_z. The default internal variables associated with all particle states are thus:

$$\begin{aligned} Hidden = \quad & \{stage,\, stage_{new},\, count,\, count_{new}\} \cup \\ & \bigcup_{c \in Chem} \{c_{release}\,\} \\ ReadOnly = \quad & \bigcup_{c \in Chem} \{c_{uptake},\, c'_{uptake},\,\} \cup \\ & \bigcup_{t \in VIter} \{t_{ingest},\, t'_{ingest},\,\} \cup \end{aligned}$$

The internal vectors $c'_{uptake}, c'_{release}$ will again have associated history index sets.

For each variety iterator $t \in VIter_i$, there is an associated ingest vector which is used to store the number of individuals of each target variety/stage specified in t. The ingest vectors are maintained automatically, and give the number of individuals of each variety that the particle ingested in the previous timestep, or a timestep in the history represented by t'_{ingest}. This ingest vector is automatically created by Planktonica with the same arity as the iterator vector with which it is associated.

As an implementation note, the ingestion vector contains only pointers to a single vector for each particle, which records the number of individuals of each variety that the particle ingested in the previous timestep. The ingest variables also have an associated

history, t'_{ingest}.

Again, recall that state variables (those in both $State_i$ and $VState_i$ above) have associated histories. Thus, define

$$\begin{aligned} State'_i &= \cup_{s \in State_i} \{ s, s' \} \cup \{ z, z' \} \\ VState'_i &= \cup_{v \in VState_i} \{ v, v' \} \end{aligned}$$

where the valid indices of s' and v' are given by their respective index sets. For variety v, therefore, the set of internal identifiers associated with v[1] is given by

$$\begin{aligned} PVars_v = \; & Hidden \cup ReadOnly \cup Ing_v \cup State'_v \cup Local_v \cup \\ & VState'_v \cup VIter_v \cup Const_v \cup VLocal_v \cup VConst_v \end{aligned}$$

It is also convenient to define universal sets $State$, $Local$, $Const$, $VIter$, $VState$, $VLocal$, $VConst$, taken over all varieties. For example $State = \cup_{1 \le i \le N_v} State'_i$, $Local = \cup_{1 \le i \le N_v} Local_i$, and so on. Recall that members of $VState$, $VLocal$ and $VConst$ will be explicitly linked to some variety iterator in $VIter$; this is enforced when the variables are created.

Each particle of type v in the simulation has an associated state which is represented in the rewrite rules as a function of type $PState_v = PVars_v \rightarrow Double$, i.e. a function that maps an identifier ($\in PVars_v$) to its value.

Although the different particle varieties are treated separately here with respect to their state it is convenient to define a particle state *superclass*, $PState$, to which each $PState_i, 1 \le i \le N_v$, belongs[2]. The state of all particles in a simulation can then be considered collectively in the form of a single particle state list of type $[PState]$.

The turbocline depth is computed by the physics code behind the curtain at the beginning (equivalently the end) of each timestep. In the rules that follow the turbocline depth parameter will consistently be named δ. It corresponds to the predefined variable $MLDepth$ that is referred to by the user.

[1] This means that the variables identified will have a binding in the internal state of each particle that is an instance of variety v.

[2] The Planktonica interface ensures that the rules for one variety cannot illegally access variables associated with another.

Finally, note that the following global constants are also defined:

h	The chemistry grid spacing in metres
L	The number of grid points in the chemistry grid equivalently the number of layers
$\triangle t$	The timestep size in hours

5.4 Rewrite Rules: Biology

The operational semantics is defined by rewrite rules. The objective is to describe the way in which the state of each particle is updated via the execution of the various rules associated with it, or more specifically, the rules associated with the variety of which it is an instance, and its stage. It is important to understand that a considerable amount of work is also performed between timesteps. However, as this is performed behind the curtain, we focus on formalising the state updates that are under direct control of the user. These updates apply exclusively to the internal state of the various particles, although new particles may be formed (by calls to *pchange* and *create*). They may in turn cause changes in both chemistry and physics variables, but these updates take place only between timesteps and behind the curtain. These will be described in Section 5.6.

In what follows, if v is a variety-based state variable then $|v|$ will be used to denote its arity – equivalent to the length of the variety iterator to which it is linked.

5.4.1 Particle List Update

Given the mixed layer depth (δ), the state of the physics and chemistry grids, i.e. the physics and chemistry environments ρ_{phys} and ρ_{chem}, and the list of particle states, the function $U^* : Depth \rightarrow PEnv \rightarrow CEnv \rightarrow [PState] \rightarrow ([PState], [PState])$ defines the particle updates in terms of an individual particle update function $U : PEnv \rightarrow CEnv \rightarrow PState \rightarrow (PState, [PState])$. The function returns the modified state of the input particles and a (possibly empty) list of new particles.

$$\begin{aligned} U^*\ \delta\ \rho_{phys}\ \rho_{chem}\ [\sigma_1, ..., \sigma_n] &= ([\sigma'_1, ..., \sigma'_n], [\nu_1, ..., \nu_n]) \\ \text{where}\quad (\sigma'_i, \nu_i) &= U\ \sigma_1\ \delta\ \rho_{phys}\ \rho_{chem} \end{aligned} \tag{5.1}$$

Note that the order in which particles are updated is unimportant.

Implementation Note: While the semantics treats the particles as a flat list, in practice data structures are used which allow efficient iteration of particles of a given stage and variety, in each layer of the column. This arrangement of data structures improves the efficiency of the various operations that occur behind the curtain between timesteps.

5.4.2 Rule Execution

The update of a single particle involves executing a sequence of rules, in the order in which they are written. This execution is defined by a function $E^* : Rules \to PState \to Depth \to PEnv \to CEnv \to ([PState], [PState])$. The rules to be executed are defined by the variety and the stage of the particle being updated. A function R (unspecified) is assumed, that returns the list of rules associated with a given variety and stage.

It is convenient to consider the rules at the syntactic level to simplify the exposition. In practice, the simulation engine executes compiled versions of each rule, but to describe what the engine does we will write the rules exactly as they appear to the user in the output from VEW Documenter.[3] We use double square brackets ($[\![$ and $]\!]$) to delimit syntactic objects. The various syntactic object types referred to are defined in Figure 5.2.

$$U\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} = E^*\ (R\,(\sigma\ var)\,(\sigma\ stage))\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \tag{5.2}$$

Recall that v' internally stores the history associated with v. The execution of a sequence of rules is defined in terms of a function $E : Rule \to PState \to Depth \to PEnv \to CEnv \to (PState, [PState])$ for processing one such rule.

$$\begin{aligned} &E^*\ [\![R_1; ...; R_n;]\!]\ \sigma_0\ \delta\ \rho_{phys}\ \rho_{chem} = (\sigma_n, [\nu_1, ..., \nu_n]) \\ &\text{where} \quad (\sigma_i, \eta_i\ ) = \ E\ [\![R_i]\!]\ \sigma_{i-1}\ \delta\ \rho_{phys}\ \rho_{chem},\ 1 \le i \le n \end{aligned} \tag{5.3}$$

[3]This will ultimately be how the rules appear in the Planktonica user interface. At present, as previously discussed, the equations input by the user are rendered slightly differently to the way they are shown here.

Note that the rules are invoked in order with the (possibly) modified particle state being passed down from one rule to the next. Each rule may result in a new particle being created (via the *create* and *pchange* functions). The result therefore includes a list of new particles (particle states) which will be either empty ([]) or singleton for each call to E.

Note: In the rules that follow it is important to remember that the terms in double square brackets denote syntactic terms. A variable in the right hand-side of a rule refers to a variable in the model state, which is itself represented by the physics and chemistry environments, and the various particle states.

5.4.3 Assignments

Direct Assignments can be carried out on scalar state variables, scalar local variables and their variety-based equivalents.

An assignment of an expression e to a variable v with an associated history, causes $v \downarrow 0$ to be assigned the result of evaluating e. Recall that $v \downarrow 0$ is the value that will be used to update v for the next time step. No new particles are created.

If σ represents the state of a particle then $\sigma\{x := v\}$ denotes the state σ updated in such a way that $(\sigma\{x := v\})\ x\ = v$. Thus, for $v\ \in\ (\{\, z \,\} \cup State)$:

$$E\,[\![v = e]\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} = (\sigma\ \{v' \downarrow 0 := \varepsilon\,[\![e]\!]\ \sigma\ (\sigma\ z)\ \delta\ \rho_{phys}\ \rho_{chem}\}, [\,]) \quad (5.4)$$

The function $\varepsilon : Expr \rightarrow PState \rightarrow Depth \rightarrow Depth \rightarrow PEnv \rightarrow CEnv \rightarrow Double$ computes the value of a given expression given the particle state, its depth, the mixing layer depth and the physics and chemistry environments. The particle depth is a separate parameter as the particle may pass through several layers in the same timestep. The binding for z (depth) in the particle state (i.e. $(\sigma\ z)$ above) represents the depth of the particle in the current timestep.

An assignment may be performed on a local variable w, in which case w assumes

the value of e immediately. Thus for $w \in Local$:

$$\begin{aligned} & E\,[\![w = e]\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \\ & = (\sigma\ \{w := (\varepsilon\,[\![e]\!]\ \sigma\ (\sigma\ z)\ \delta\ \rho_{phys}\ \rho_{chem})\}, [\,]) \end{aligned} \tag{5.5}$$

An assignment to a variety-based state variable $v \in VState$ requires an iteration for each element of v. For each iteration, $i, 1 \leq i \leq |v|$, the value for $(v' \downarrow i) \downarrow 0$ (the value the i^{th} element of v will assume in the next timestep) is computed by the function $\overline{\varepsilon}$. Recall that variables in $VState$ are represented internally as two-dimensional structures (each as a vector of vectors). The function $\overline{\varepsilon}$ differs from ε in that any variety-based variables will be indexed using the additional parameter (i above). Thus, for $v \in VState$:

$$\begin{aligned} & E\,[\![v = e]\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \\ & = (\sigma\ \{(v' \downarrow i) \downarrow 0 := (\overline{\varepsilon}\,[\![e]\!]\,i\ \sigma\ (\sigma\ z)\ \delta\ \rho_{phys}\ \rho_{chem}), 1 \leq i \leq |v|\}, [\,]) \end{aligned} \tag{5.6}$$

An assignment to a local variety-based variable w requires a similar iteration, with the results being immediately applied to each element. For $w \in VLocal$:

$$\begin{aligned} & E\,[\![w = e]\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \\ & = (\sigma\ \{w \downarrow i := (\overline{\varepsilon}\,[\![e]\!]\,i\ \sigma\ (\sigma\ z)\ \delta\ \rho_{phys}\ \rho_{chem}), 1 \leq i \leq |w|\}, [\,]) \end{aligned} \tag{5.7}$$

5.4.4 Differential Assignments

Assignments to state variables can also be written in differential form, where the right-hand side defines a rate of change. At present a straightforward Eulerian integration step is performed. For $v \in State$:

$$\begin{aligned} & E\left[\!\!\left[\frac{dv}{dt} = e\right]\!\!\right]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \\ & = (\sigma\{v' \downarrow 0 := v + \triangle t\ \ (\varepsilon\,[\![e]\!]\ \sigma\ (\sigma\ z)\ \delta\ \rho_{phys}\ \rho_{chem})\}, [\,]) \end{aligned} \tag{5.8}$$

A similar rule applies for variety-based variables, but recall that the internal vector v' is two-dimensional. Thus, for $v \in VState$:

$$\begin{aligned} & E\left[\!\!\left[\frac{dv}{dt} = e\right]\!\!\right]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \\ & = (\sigma\{(v' \downarrow i) \downarrow 0 := v \downarrow 0 + \triangle t\ \ (\overline{\varepsilon}\,[\![e]\!]\,i\ \sigma\ (\sigma\ z)\ \delta\ \rho_{phys}\ \rho_{chem}), 0 < i \leq |v|\}, [\,]) \end{aligned} \tag{5.9}$$

5.4.5 Conditional Execution

$$\begin{aligned} & E\,[\![if\ B\ then\ R\ else\ R']\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \\ & = \textbf{if}\ \beta\,[\![B]\!]\ \sigma\ (\sigma\, z)\ \delta\ \rho_{phys}\ \rho_{chem} = tt \quad \textbf{then} \quad E\,[\![R]\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else} \quad E\,[\![R']\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \end{aligned} \tag{5.10}$$

The boolean function $\beta : BExpr \to PState \to Depth \to Depth \to PEnv \to CEnv \to \{tt, ff\}$ computes either tt (true) or ff (false).

5.4.6 Chemical Uptake

A request to uptake an amount a (in micrograms) of a chemical $c \in Chem$ causes the uptake variable for c to be increased by a. Several uptakes may occur for the same chemical during a timestep, in which case they are accumulated. Thus,

$$\begin{aligned} & E\,[\![uptake(c,e)]\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \\ & = (\sigma\{c'_{uptake} \downarrow 0 := c_{uptake} + (\varepsilon\,[\![e]\!]\ \sigma\ (\sigma\, z)\ \delta\ \rho_{phys}\ \rho_{chem})\}, [\,]) \end{aligned} \tag{5.11}$$

Note that the condition that $c \in Chem$ is enforced by Planktonica when the rule is entered.

5.4.7 Chemical Release

Chemical release operates similarly to uptake:

$$\begin{aligned} & E\,[\![release(c,e)]\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} \\ & = \sigma\{c'_{release} \downarrow 0 := c_{release} + (\varepsilon\,[\![e]\!]\ \sigma\ (\sigma\, z)\ \delta\ \rho_{phys}\ \rho_{chem})\}, [\,]) \end{aligned} \tag{5.12}$$

5.4.8 Ingestion

For a particle that ingests members of a target set (specified by variety and growth stage), the ingestion is defined to take place over a trajectory defined by the depths of the particle in the current and previous time steps, i.e. z and $(z \downarrow\text{-}1)$ respectively. The first argument of *ingest* is always a variety iterator. The second argument is an expression for computing the ingestion rate. This expression is computed for each element index in the variety iterator (forced iteration). If the expression refers to a

variety-based variable (these will always be linked to the variety iterator), this variable will be indexed accordingly.

In the following rewrite rule z_{max} and z_{min} define the two extremes of the trajectory. For the purposes of ingestion, the direction of travel is not important. It is important to note that the ingesting particle only moves through a proportion of the topmost and bottommost layer the distances p_{top} and p_{bot}, but travels through the entirety of each intermediate layer - see Figure 3.4.4.

The ambient environment of the particle changes as it moves, hence the physical and chemical environments ρ_{phys} and ρ_{chem} are indexed with the depth of the particle along its trajectory. The number of individuals of each target type ingested along the trajectory is delivered in the variable x_{ingest}; this is linked to the target variety iteration variable x as previously described. Thus, for $x \in VIter$,

$$
\begin{aligned}
&E\,[\![ingest(x,r)]\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} = \\
&(\sigma\{(x_{ingest} \downarrow i) \downarrow 0 := G_i, 1 \leq i \leq |x|\}, [\,]) \\
&\quad \text{where } G_i = \tfrac{s_h \times \triangle t}{z_{max} - z_{min}} \times (top_i + mid_i + bot_i) \quad \text{where} \\
&\qquad top_i = p_{top}\ \overline{\varepsilon}\,[\![r]\!]\ i\ \sigma\ z_{max}\ \delta\ \rho_{phys}\ \rho_{chem} \\
&\qquad bot_i = p_{bot}\ \overline{\varepsilon}\,[\![r]\!]\ i\ \sigma\ z_{min}\ \delta\ \rho_{phys}\ \rho_{chem} \\
&\qquad mid_i = \textstyle\sum_{l=l_{min}+1}^{l_{max}-1} \overline{\varepsilon}\,[\![r]\!]\ i\ \sigma\ (l\ h)\ \delta\ \rho_{phys}\ \rho_{chem} \\
&\qquad p_{top} = l_{min} - \tfrac{z_{min}}{h} \\
&\qquad p_{bot} = \tfrac{z_{max}}{h} - l_{max} \\
&\qquad z_{max} = max\,(\sigma\ z,\ \sigma\ (z \downarrow \text{-}1)) \\
&\qquad z_{min} = min\,(\sigma\ z,\ \sigma\ (z \downarrow \text{-}1)) \\
&\qquad l_{min} = \lfloor \tfrac{z_{min}}{h} \rfloor \\
&\qquad l_{max} = \lfloor \tfrac{z_{max}}{h} \rfloor
\end{aligned}
\tag{5.13}
$$

5.4.9 Stage Changes

Stage changes are straightforward. Recall the function S which maps growth stage names to stage numbers for a given variety.

$$
E\,[\![change(s)]\!]\ \sigma\ \ \delta\ \rho_{phys}\ \rho_{chem} = (\sigma\{stage_{new} := S\ (\sigma\ var)\ s\}, [\,]) \tag{5.14}
$$

Proportional stage changes cause a new particle to be created whose cell count is the stated proportion of that of the parent. The parent cell count is reduced accordingly. The new particle (state) is formed by *cloning* the parent prior to adjusting the cell count. A function *clone* is assumed which makes a copy of a given particle state.

$$\begin{aligned}
&E\,[\![pchange(p,s)]\!]\ \sigma\ \delta\ \rho_{phys}\ \rho_{chem} = \\
&(\ \sigma\{count_{new} := c\}, [\ (clone\ \sigma)\{count_{new} := c'\}\ ]\ ) \\
&\quad c \ = \ count \times (1 - (\varepsilon\,[\![p]\!]\,\sigma\ (\sigma\ z)\ \delta\ \rho_{phys}\ \rho_{chem})) \\
&\quad c' \ = \ count \times (\varepsilon\,[\![p]\!]\,\sigma\ (\sigma\ z)\ \delta\ \rho_{phys}\ \rho_{chem}))
\end{aligned} \tag{5.15}$$

5.4.10 Particle Division

The *divide* function causes the sub-population size to be multiplied by specified amount. No new particles are created.

$$\begin{aligned}
&E\,[\![divide(e)]\!]\ \sigma\ \ \delta\ \rho_{phys}\ \rho_{chem} = \\
&(\sigma\{count_{new} := count \times (\varepsilon\,[\![e]\!]\,\sigma\ (\sigma\ z)\ \delta\ \rho_{phys}\ \rho_{chem})\}, [\,])
\end{aligned} \tag{5.16}$$

5.4.11 Particle Creation

Recall from Section 3.4.1 that particle creation causes a clone of the parent to be created, with a specified sub-population and stage. The new particle may optionally be modified by a series of assignments. In this case the assignments are effected by the function E^*. The assignments cannot create new particles so the second component returned by E^* will be $[\,]$.

$$\begin{aligned}
&E\,[\![create\,(x,s)]\!]\,\sigma\ \delta\ \rho_{phys}\ \rho_{chem} \\
&= (\sigma, [\ (clone\ \sigma)\{\ stage_{new} := S\ (\sigma\ var)\ s, \\
&\qquad\qquad count_{new} := \varepsilon\,[\![x]\!]\,\sigma\ (\sigma\ z)\ \delta\ \rho_{phys}\ \rho_{chem}\}\ ])
\end{aligned} \tag{5.17}$$

$$
\begin{aligned}
&E\,[\![create\,(x,s)\ \ with\ \{v_1 = e_1, ..., v_n = e_n\}]\!]\ \ \sigma\ \ \delta\ \rho_{phys}\ \rho_{chem} \\
&= (\sigma, [\sigma']) \\
&\quad \text{where}\ \ (\sigma', \nu) = E^*[\![v_1 = e_1; ..., v_n = e_n;]\!]\ \sigma''\ \delta\ \rho_{phys}\ \rho_{chem} \\
&\qquad\qquad \sigma'' = (clone\ \sigma)\{ \\
&\qquad\qquad\quad stage_{new} := S(\sigma\, var)\, s, \\
&\qquad\qquad\quad count_{new} := \varepsilon\,[\![x]\!]\,\sigma\ (\sigma\, z)\ \delta\ \rho_{phys}\ \rho_{chem}\}
\end{aligned}
\tag{5.18}
$$

5.4.12 Expression Evaluation

The function $\varepsilon : PState \to Depth \to Depth \to PEnv \to CEnv \to Double$ computes the value of an expression with scalar type, as previously described. It is defined in terms of a similar function $\overline{\varepsilon}$ which is used to implement forced iteration. $\overline{\varepsilon}$ takes an additional argument, the variety index (type Int), which is used to index any variety-based variables in the given expression. In the case of scalar expression evaluation, where there is no iteration at the topmost level, an index of 0 is passed to $\overline{\varepsilon}$. Thus:

$$\varepsilon\ e\ = \overline{\varepsilon}\ e\ 0 \tag{5.19}$$

The rules for $\overline{\varepsilon}$ now follow. Note:

- The particle depth is defined by an additional parameter (ψ), rather than by $\sigma\, z$ as the expression may be being computed along a trajectory.
- Particles may only access the concentration variables $c_{conc}, c \in Chem$ in the chemistry grid. The other chemistry variables may only be accessed by the corresponding chemistry rules (see Section 5.5).
- The only variables that can be indexed by history are state variables ($State$ and $VState$) including the particle depth (z), the chemical uptake variables for each chemical c_{uptake}, and the ingestion variables, t_{ingest}; the auxiliary function H performs history indexing. The index is an expression; when computed it is rounded down to the nearest integer.

- A reference to a variety-based variable (vector) causes the corresponding vector to be indexed (indexing operator $\downarrow$). Note that indexing a vector at 0 would correspond to an error, but the Planktonica user interface ensures that scalar expressions contain no vector references at the topmost level[4]; in this sense all models are correctly typed.

- Condition expression evaluation is defined in terms of the boolean function $\overline{\beta}$. This is analogous to $\overline{\varepsilon}$ in that it carries an integer index that is used to index any variety-based vector variable.

- The three vector operators ($vAvg$, $vMul$ and $vSum$) reduce vectors to scalars. The function *arity* (unspecified) computes the arity of a given expression, i.e. the arity of its component vector(s). Note that all vectors appearing at the top level in an expression have the same arity by construction as they must all be linked to a common variety iterator. The arity of a scalar expression is defined to be 0, in which case the three functions treat the scalar as though it were a vector with a single element. Thus, for example, $vAvg(e)$ is defined to be e if e is scalar.

- The rule for *integrate* is similar in structure to that of *ingest*. For a particle with trajectory (z_{min}, z_{max}) the expression $integrate(e)$ is interpreted as

$$\int_{z_{min}}^{z_{max}} f(z)\, dz$$

  where $f(z) = e$. The integration is, of course, formed by a finite sum because of the discretisation of the water column.

- The basic mathematical operators have their usual meaning, so the rules are omitted. Recall, however, that juxtaposition means multiplication. The usual mathematical associativity and precedence rules apply.

$$\overline{\varepsilon}\, [\![x]\!]\, i\ \sigma\ \psi\ \delta\ \rho_{phys}\ \rho_{chem} = x, \quad x \in Num \tag{5.20}$$

[4] A scalar-valued expression may, of course, make use of the vector operations $vAvg, vMul$ and $vSum$ but their vector arguments will not be at the topmost level.

$$\overline{\varepsilon}\,[\![MLDepth]\!]\,i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem} = \delta \tag{5.21}$$

$$\overline{\varepsilon}\,[\![v]\!]\,i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem} = \rho_{phys}\;\psi\;v, \quad v \in PhVars \tag{5.22}$$

$$\overline{\varepsilon}\,[\![c_{conc}]\!]\,i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem} = (\rho_{chem}\,@\,\psi)\,c_{conc}, \quad c \in Chem \tag{5.23}$$

$$\begin{aligned}&\overline{\varepsilon}\left[\!\left[v_{[e]}\right]\!\right]\,i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}\\ &= H[\![v]\!]\;\lfloor\overline{\varepsilon}\,[\![e]\!]\,i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}\rfloor\;\sigma, \quad v \in (Depth \cup State \cup VState)\end{aligned} \tag{5.24}$$

$$\overline{\varepsilon}\,[\![z]\!]\,i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem} = \psi \tag{5.25}$$

$$\overline{\varepsilon}\,[\![v]\!]\,i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem} = \sigma\,v, \quad v \in (State \cup Local \cup Const) \tag{5.26}$$

$$\begin{aligned}&\overline{\varepsilon}\,[\![v]\!]\,i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}\\ &= (\sigma\,v)\downarrow i, \quad v \in (VIter \cup VState \cup VLocal \cup VConst\end{aligned} \tag{5.27}$$

$$\begin{aligned}&\overline{\varepsilon}\,[\![vAvg(e)]\!]\;\;i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}\\ &= \textbf{if}\;arity(e) = 0 \quad \textbf{then} \quad \varepsilon\,[\![e]\!]\;\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem},\\ &\qquad\qquad\;\; \textbf{else} \quad \tfrac{\varepsilon[\![vSum(e)]\!]\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}}{arity(e)}\end{aligned} \tag{5.28}$$

$$\begin{aligned}&\overline{\varepsilon}\,[\![vMul(e)]\!]\;\;i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}\\ &= \textbf{if}\;arity(e) = 0 \quad \textbf{then} \quad \varepsilon\,[\![e]\!]\;\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem},\\ &\qquad\qquad\;\; \textbf{else} \quad \textstyle\prod_{j=1}^{arity(e)}\varepsilon\,[\![e]\!]\;\;j\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}\end{aligned} \tag{5.29}$$

$$\begin{aligned}&\overline{\varepsilon}\,[\![vSum(e)]\!]\;\;i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}\\ &= \textbf{if}\;arity(e) = 0 \quad \textbf{then} \quad \varepsilon\,[\![e]\!]\;j\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}\\ &\qquad\qquad\;\; \textbf{else} \quad \textstyle\sum_{j=1}^{arity(e)}\varepsilon\,[\![e]\!]\;\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}\end{aligned} \tag{5.30}$$

$$\begin{aligned}&\overline{\varepsilon}\,[\![if\;B\;then\;E\;else\;E']\!]\;\;i\;\sigma\;\delta\;\rho_{phys}\;\rho_{chem}\\ &= \textbf{if}\;\overline{\beta}\,[\![B]\!]\;\;i\;\sigma\;(\sigma\,z)\;\delta\;\rho_{phys}\;\rho_{chem} = tt \quad \textbf{then} \quad \overline{\varepsilon}\,[\![R]\!]\;\;i\;\sigma\;\delta\;\rho_{phys}\;\rho_{chem}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\;\; \textbf{else} \quad \overline{\varepsilon}\,[\![R']\!]\;\;i\;\sigma\;\delta\;\rho_{phys}\;\rho_{chem}\end{aligned} \tag{5.31}$$

$$
\begin{aligned}
&\overline{\varepsilon}\,[\![integrate(e)]\!]\ i\ \sigma\ \psi\ \delta\ \rho_{phys}\ \rho_{chem} \\
&= \tfrac{s_h \times \triangle t}{z_{max} - z_{min}} \times (top_i + mid_i + bot_i) \\
&\quad \text{where}\ \ top_i = p_{top}\ \varepsilon\,[\![r]\!]\ \sigma\ z_{max}\ \delta\ \rho_{phys}\ \rho_{chem} \\
&\qquad\qquad bot_i = p_{bot}\ \varepsilon\,[\![r]\!]\ \sigma\ z_{min}\ \delta\ \rho_{phys}\ \rho_{chem} \\
&\qquad\qquad mid_i = \textstyle\sum_{l=l_{min}+1}^{l_{max}-1} \varepsilon\,[\![r]\!]\ \sigma\ (l\ h)\ \delta\ \rho_{phys}\ \rho_{chem} \\
&\qquad\qquad p_{top} = l_{min} - \tfrac{z_{min}}{h} \\
&\qquad\qquad p_{bot} = \tfrac{z_{max}}{h} - l_{max} \\
&\qquad\qquad z_{max} = max\,(\sigma\ z,\ \sigma\ (z{\downarrow}\text{-}1)) \\
&\qquad\qquad z_{min} = min\,(\sigma\ z,\ \sigma\ (z{\downarrow}\text{-}1)) \\
&\qquad\qquad l_{min} = \lfloor \tfrac{z_{min}}{h} \rfloor \\
&\qquad\qquad l_{max} = \lfloor \tfrac{z_{max}}{h} \rfloor
\end{aligned}
\tag{5.32}
$$

History Indexing

$$H[\![v]\!]\ n\ \sigma = (\sigma\ v')\downarrow n \tag{5.33}$$

5.4.13 Boolean Expression Evaluation

Boolean values (tt and ff) are not first-class objects in the modelling language, but they are computed as part of both conditional statement and conditional expression evaluation. Similar to numerical expressions, there are non-indexed and indexed variants, analogous to ε and $\overline{\varepsilon}$.

The function β is used only in conditional statement evaluation where there can be no forced iteration (iteration cannot be induced by the predicate evaluation, although the rules in each branch may do so). Thus,

$$\beta\ b = \overline{\beta}\ b\ 0 \tag{5.34}$$

The function β is defined in terms of the function $\overline{\beta}$. The conventional comparison operators ($=, \neq, <, >, \leq, \geq$) have their usual interpretation, so the corresponding rules for $\overline{\beta}$ are omitted. For completeness, we include the definitions of $all, some$ and $none$. Note that an attempt to evaluate these expressions with an index *other than* 0 would

correspond to an error. Again, however, the Planktonica input system enforces correctness in this sense: the variety-based variable reference is not permitted in this context as this would imply a forced iteration at some outer level.

5.4.14 Boolean Expressions for Statements

$$\overline{\beta}\,[\![all(b)]\!]\;\; i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem} = \bigwedge_{1\leq j\leq arity(b)} \overline{\beta}\,[\![b]\!]\;j\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem} \quad (5.35)$$

$$\overline{\beta}\,[\![some(b)]\!]\;\; i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem} = \bigvee_{1\leq j\leq arity(b)} \overline{\beta}\,[\![b]\!]\;j\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem} \quad (5.36)$$

$$\overline{\beta}\,[\![none(b)]\!]\;\; i\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem} = !\;(\beta\,[\![some(b)]\!]\;\sigma\;\psi\;\delta\;\rho_{phys}\;\rho_{chem}) \quad (5.37)$$

5.5 Rewrite Rules: Chemistry

Analogous to the "update" code for each variety (functional group) and growth stage, there is also update code associated with each chemical. There are no particles associated with chemistry, however: chemical variables are associated with grid points rather than particles. In the top-level update loop (equivalent to U^* in the biology) each user-defined chemical in each layer of the chemistry grid is updated in turn.

The rewrite rules for the chemistry modelling language are not listed as they are a subset of those already presented for the biology. The differences are:

- State variables apply to a chemistry grid point rather than a particle
- There are no rules associated with variety-based variables ($VIter$, $VConst$, $VLocal$ and $VState$).
- The metamodel support functions (*uptake*, *release*, *ingest*, *change*, *pchange*, *divide*, *create*) are not applicable.
- The variety-based functions ($vAvg$, $vSum$, $vMul$, *all*, *some* and *none*) are not applicable.

- *integrate* may not be used in expressions as this only has a meaning in the context of a particle trajectory.

5.6 Between Timesteps

Between timesteps, the following functions are performed.

- All chemical concentrations above the turbocline are averaged, as an approximation for turbulence. This causes the chemical environment ρ_{chem} to change between timesteps.

- The chemical update described in section 3.3.2 is performed. Within each layer, for each $c \in Chem$, c_{conc} is incremented by $c_{release}$ of each particle in that layer, after which the particle's $c_{release}$ is set to zero. Then, for each layer and chemical, c_{conc} in each layer is reduced by the total c_{uptake} of the particles in that layer. If c_{conc} became negative, then c_{conc} is set to zero, and the c_{uptake} of each particle is adjusted.

- The changes due to ingestion function calls are carried out. Recall that ingestion has two arguments, an iterator of populations to be ingested, and an expression defining the rate for each. For each variety in the iterator, all the particles of that type, between the predator's start and end depth, are considered, and their subpopulation sizes are reduced by the calculated rate. After all particles have performed their ingestion, if any of the sub-population sizes became negative, it is set to zero, and the particles that ingested from that sub-population are re-visited. The predators' t_{ingest} variable for that sub-population type is reduced accordingly.

- Particles are rearranged into the correct layer within the internal data structures. Particles that have dropped out of the bottom of the column, or those that have a negative depth are removed, as are those that have a sub-population size of zero.

- All variables that have a history are updated, so that their current value becomes their value in the previous timestep, and so on.

- Particle management rules are applied, which may split a number of the largest particles or merge a number of the smallest particles, causing changes in sub-population sizes of particles. See section 7.4 for details of these rules, which are defined outside of the model.

- User-defined events may cause changes to the physical environment, the chemical environment, or sub-population sizes of particles. See section 7.5.

- The physical environment is updated, using forcing climatological data, resulting in a new value of the turbocline, δ, and an updated physical environment ρ_{phys}. This also takes into account the amount of any pigments within the particles, since this affects irradiance. See the example from the WB model in section 6.3.

Chapter 6

Building The WB Model in Planktonica

In this chapter, a new Planktonica-compliant reconstruction of the WB Model in its entirety is described. The purpose is to show by example how a model is designed and constructed using Planktonica, thus justifying the need for the variable types and special functions provided.

6.1 Presentation Conventions

At present Planktonica equations and rules are entered using a very simple equation editor. The precise nature of the way equations and/or rules are input by the user is a largely cosmetic issue and is not seen as a central issue for this thesis. However, screen-shots and explanations of how certain tasks are carried out are included in appendix A.

The rules listed in this chapter are done so in a format produced by a prototype automatic documentation utility. The aim for the model building utility is that the input to Planktonica, and the documentation that might accompany a model, should be as similar as possible. The layout of rules (spacing and line breaks) is not relevant to the modelling language; this is purely aesthetic for typesetting. Similarly, within the semantics of the rule language, semicolons have been used to separate rules, but

these have been omitted in this chapter. Finally, juxtaposition of two or more variables implies multiplication throughout this chapter.

6.2 Functional Groups

The WB Model contains two functional groups: Diatoms and Copepods. Functional groups exist in one of a number of stages, which the user can define. In the WB model, stages are used to represent phases of the functional group's life-cycle, and certain functions and sub-functions are only relevant in certain stages. When creating a new function or sub-function, the user specifies the stages of the functional group in which the function operates.

Diatom Stages

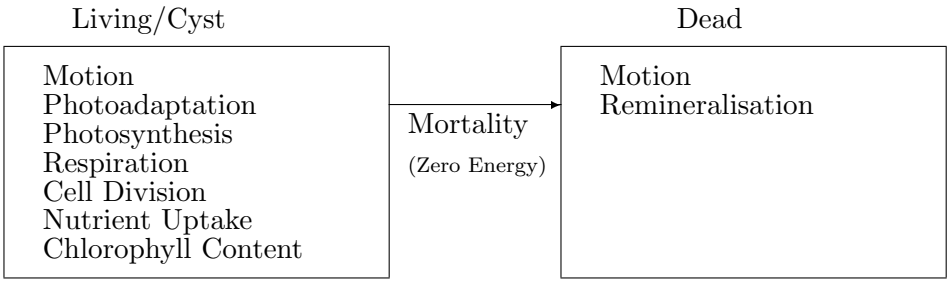

Figure 6.1: Diatom Stages

Figure 6.1 shows how diatom stages are used in the WB model. Three stages are used: living diatoms and cyst diatoms are the same except that a different respiration rule is used for diatoms in the cyst stage, during which they respire at a lower rate in the winter months. The third stage represents dead diatoms.

6.2.1 Copepod Stages

Six stages are used to represent copepods in different phases of the copepod life-cycle, shown in figure 6.2. It is created initially in a *juvenile* stage, although when reproduction occurs, the offspring are created in the *newborn* stage. Only a proportion of newborn

copepods survive to become juveniles. When a juvenile reaches a certain weight threshold it becomes an *adult*, and produces eggs. After an incubation period, the eggs hatch and the newborn offspring are created. After a specified time, the adult becomes *senile*, and after another time period they become detritus (*dead*). A sixth stage exists when a copepod has been cannibalised; in the growth function, if carbon loss by respiration outweighed carbon gained by ingestion, then the copepods cannibalise each other to make up the deficit.

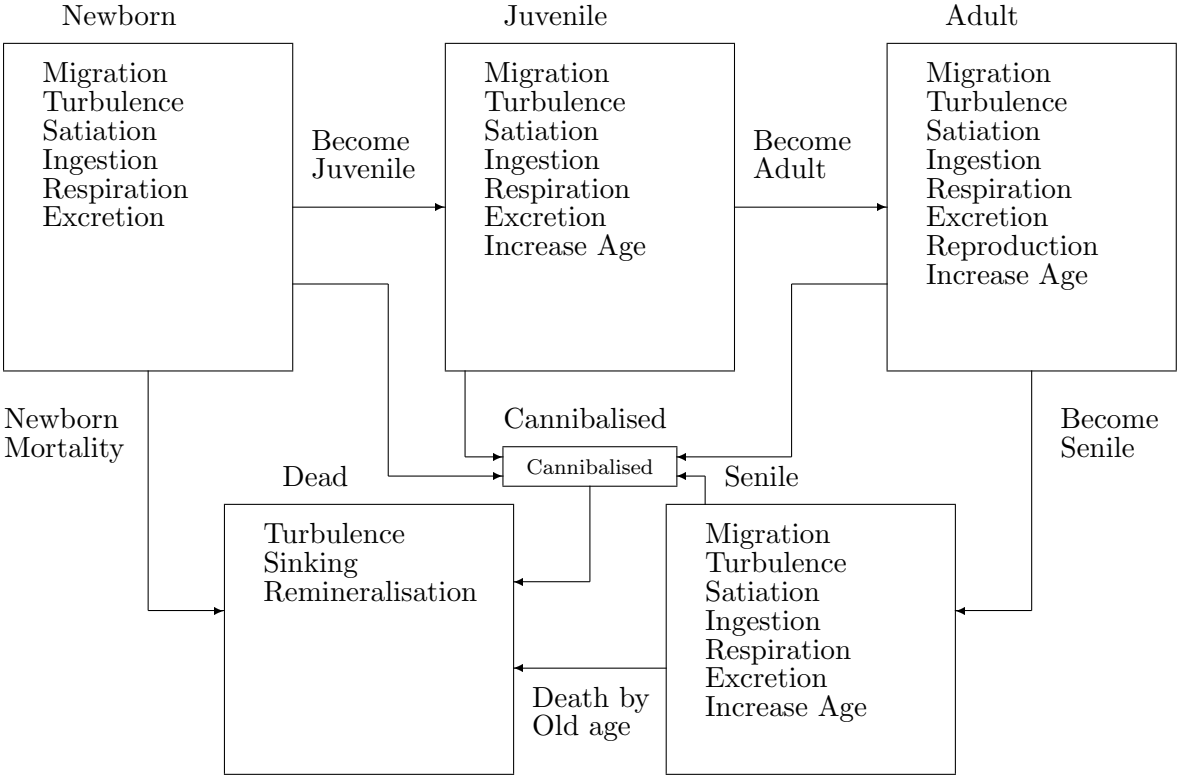

Figure 6.2: Copepod Stages

A further stage is used to represent a copepod pellet. This is actually a mis-use of the term 'stage', but being able to re-use the turbulence and sinking functions for copepods, and indeed to create the pellet itself is extremely convenient. We return to this in section 6.6.5

6.3 Chemicals and Pigments

The WB model has two chemicals and one pigment: ammonium and nitrate are absorbed by diatoms, and by photosynthesis diatoms produce the pigment chlorophyll, which behaves like a chemical except additionally has pigmentation properties, called action spectra. Each action spectra is a function of wavelength, and at present, the built-in physics module provides the irradiance through 25 wavelengths, hence this is currently the maximum resolution for the pigmentation function.

For biofeedback, two action spectra, χ and e are used. The values of these two action spectra for chlorophyll are shown for the 25 wavebands in figure 6.3. In the table, the wavelength given is the beginning of that band; for example, the value of χ is 0.121 between wavelengths 387.5nm and 412.5nm.

w (nm)	300	357.5	387.5	412.5	437.5	462.5	487.5	512.5	537.5	562.5	587.5
χ	0.0	0.0	0.121	0.109	0.095	0.077	0.061	0.047	0.041	0.035	0.035
e	1.0	1.0	0.677	0.702	0.702	0.703	0.695	0.673	0.65	0.618	0.628
w (nm)	612.5	637.5	662.5	687.5	712.5	737.5	787.5	900	1100	1300	1500
χ	0.041	0.045	0.049	0.034	0.0	0.0	0.0	0.0	0.0	0.0	0.0
e	0.65	0.672	0.687	0.62	1.0	1.0	1.0	1.0	1.0	1.0	1.0
w (nm)	1700	1900	2100	(2300)							
χ	0.0	0.0	0.0								
e	1.0	1.0	1.0								

Figure 6.3: Action spectra for chlorophyll biofeedback

The physics code calculates the irradiance as the sum of a function for each pigment:-

$$I(w) = \Sigma_p \; f\left(\chi_p(w) \; {c_p}^{e_p(w)}\right) \tag{6.1}$$

This equation is built into the kernel - it is not specified, or even seen, by the user.

The irradiance I, at a certain wavelength w is computed as a sum for each pigment, where the tables for χ and e are pigment-specific, and c_p is the concentration of that pigment within the particles in the water, a diagnostic value computed behind the curtain.

The function f represents a range of other parameters that are multiplied in the irradiance equation, which are not documented here. For details, refer to the documentation of the physics code [4].

6.4 Variable Tables

For reference, this section contains tables of all the parameters and variables used in the model.

Name	Description	Value	Units
d_{year}	Day of year (section 6.5.3)	-	$none$
$MLDepth$	Turbocline depth	-	m
$\triangle t$	Timestep size	0.5	h
π	Constant pi	π	none
s_h	Convert from seconds to hours	3600	$s^{-1}h$

Figure 6.4: Global Identifiers

Name	Description	Units
A_{conc}	Ammonium concentration	$\mu g\ Nitrogen\ m^{-3}$
C_{conc}	Carbon concentration	$\mu g\ Carbon\ m^{-3}$
Chl_{conc}	Chlorophyll concentration	$\mu g\ Chlorophyll\ m^{-3}$
I_v	Visible Irradiance	Wm^{-2}
N_{conc}	Nitrate concentration	$\mu g\ Nitrogen\ m^{-3}$
T	Temperature	$^{\circ}C$

Figure 6.5: Ambient Environmental Variables

Name	Description	Units
A_{ingest}	Ammonium ingested last timestep	$\mu g\ Nitrogen$
A_{pool}	Ammonium pool	$\mu g\ Nitrogen$
A_{uptake}	Ammonium uptake last timestep	$\mu g\ Nitrogen$
C_{ingest}	Carbon ingested last timestep	$\mu g\ Carbon$
C_{pool}	Carbon pool	$\mu g\ Carbon$
Chl_{pool}	Chlorophyll pool	$\mu g\ Chlorophyll$
N_{ingest}	Nitrate ingested last timestep	$\mu g\ Nitrogen$
N_{pool}	Nitrate pool	$\mu g\ Nitrogen$
N_{uptake}	Nitrate uptake last timestep	$\mu g\ Nitrogen$
z	Depth	m

Figure 6.6: State Variables for all particles

Note that for copepods there is also a single variety iterator, $\overline{P}^*$, which the vectors $\overline{F}$, $\overline{k_I}$, $\overline{P_{min}}$, $\overline{I_{gmax}}$, $\overline{I_{gv}}$, $\overline{s_1}$ and $\overline{s_2}$ are associated with. For the WB Model, the iterator $\overline{P}^*$ contains two elements, which are members of the default diatom variety (there is only one variety of diatom in the WB model), in their *live* and *cyst* stage. Ingestion is assumed to behave similarly for the two stages, hence the members of each variety-based

Name	Description	Value	Units
A_F	Radius of cell	1x10−5	m
A_{remin}	Ammonium remineralisation rate	0.0020833	$\mu g\ Nitrogen\ h^{-1}$
d_{cyst}	Day of year in which to enter cyst stage	305	none
d_{decyst}	Day of year in which to leave cyst stage	31	none
E_c	Energy required for cell division	$0.14\text{x}10^{-3}$	J
E_{max}	Maximum carbon pool (energy)	2.66x10−4	J
k_A	Half saturation for ammonium	0.5	$\mu g\ Nitrogen\ m^{-3}$
k_c	Carbon content of a diatom	$4.6\text{x}10^{-4}$	$\mu g\ Carbon$
k_{chl}	Chlorophyll content of a diatom	$9.2\text{x}10^{-10}$	$\mu g\ Chlorophyll$
k_F	Light absorbtion parameter	0.63	none
k_N	Half saturation for nitrate	0.5	$\mu g\ Nitrogen\ m^{-3}$
N_c	Nitrogen required for cell division	$4\text{x}10^{-9}$	$\mu g\ Nitrogen$
N_{pmax}	Maximum nitrogen pool	1.16x10−8	$\mu g\ Nitrogen$
N_{remin}	Nitrate remineralisation Rate	0.0020833	$\mu g\ Nitrogen\ h^{-1}$
R_C	Respiration parameter for cyst diatom	$1\text{x}10^{-7}$	Jh^{-1}
R_L	Respiration parameter for living diatom	$2\text{x}10^{-7}$	Jh^{-1}
t_a	Adaptation time scale	5	h
T_r	Reference temperature	10	$^{\circ}C$
u_A	Maximum uptake of ammonium	$4\text{x}10^{-10}$	$\mu g\ Nitrogen\ h^{-1}$
u_N	Maximum uptake of nitrate	$4\text{x}10^{-10}$	$\mu g\ Nitrogen\ h^{-1}$
V_p	Sinking rate	0.004167	mh^{-1}

Figure 6.7: Constant Parameters for Diatoms

parameter are assumed to be the same, and a single value is listed in the tables above.

Name	Description	Init Value	Units
E_p	Energy Pool	1x10−4	J
I_m	Photoadaptive Variable	10	Wm^{-2}

Figure 6.8: State Variables for Diatoms

Name	Description	Units
A_{cdiv}	Ammonium change by cell division	$\mu g\ Nitrogen$
c_{div}	Cell division flag	$none$
dA	Change in Ammonium	$\mu g\ Nitrogen$
dN	Change in Nitrate	$\mu g\ Nitrogen$
Ec_{cdiv}	Energy change by cell division	Jh^{-1}
$Ec_{estimate}$	Estimated new energy	Jh^{-1}
Ec_{max}	Maximum energy pool	J
Ec_{photo}	Energy change by photosynthesis	Jh^{-1}
Ec_{resp}	Energy change by respiration	Jh^{-1}
N_{cdiv}	Nitrate change by cell division	$\mu g\ Nitrogen$
$Nitrogen$	Total Nitrogen	$\mu g\ Nitrogen$
n_{red}	Flag for reducing nitrogen in cell division	$none$
s	Scaling factor for uptake	$none$
z_{sink}	Displacement by sinking	m
z_{turb}	Displacement by turbulence	m

Figure 6.9: Local/Exported Variables for Diatoms

Name	Description	Value	Units
A_I	Incubation time	480	h
A_{remin}	Ammonium remineralisation rate	0.0020833	$\mu g\ Nitrogen\ h^{-1}$
$\overline{F}$	Filtration rate	1x10^{-9}	$m^3 s^{-1}$
G_{max}	Weight required for maturity	100	$\mu g\ Carbon$
G_{min}	Body weight at birth	0.2	$\mu g\ Carbon$
I_{mp}	Probability of infant mortality	0.9	none
I_r	Reference isolume	1	Wm^{-2}
k_a	Assimilation parameter	1	none
k_b	Background respiration parameter	0.1	none
k_cp	Carbon content per diatom	4.6x10^{-4}	$\mu g\ Carbon\ plankton^{-1}$
$\overline{k_I}$	Half-saturation constant	4x10^6	$plankton\ m^{-3}$
L	Maximum life time of adult	960	h
N_{remin}	Nitrate remineralisation rate	0.0020833	$\mu g\ Nitrogen\ h^{-1}$
$\overline{P_{min}}$	Minimum threshold for ingestion	100000	$plankton\ m^{-3}$
r_{ac}	Ammonium to carbon ratio	8.7x10^{-6}	$\mu g\ Nitrogen \mu g\ Carbon^{-1}$
R_b	Basal respiration parameter	0.3x10^{-3}	h^{-1}
R_s	Assimilation parameter	0.3	none
$\overline{s_1}$	Ingestion Parameter	4.2	$plankton\ s^{-1}$
$\overline{s_2}$	Ingestion Parameter	3.2	$plankton\ s^{-1}$
S_{min}	Min. satiation before course alteration	0	none
t_m	Relaxation time	4	h
T_r	Reference temperature	10	$^\circ C$
V_{max}	Maximum swimming speed	45	mh^{-1}
V_z	Sinking rate	0.004167	mh^{-1}

Figure 6.10: Constant Parameters for Copepods

Name	Description	Init Value	Units
A_r	Age since maturity	0	h
I_t	Target isolume	0	Wm^{-2}
$\overline{I_{gmax}}$	Maximum ingestion rate	0	$plankton\ s^{-1}$
R	Total respiration rate	1	$\mu g\ Carbon\ h^{-1}$
R_{ass}	Respiration cost of food assimilation	1	$\mu g\ Carbon\ h^{-1}$
R_{bac}	Background Respiration Rate	0	$\mu g\ Carbon\ h^{-1}$
R_{bas}	Basal Respiration Rate	0	$\mu g\ Carbon\ h^{-1}$
S	Satiation	0.5	none

Figure 6.11: State Variables for Copepods

Name	Description	Units
C_{Ass}	Carbon Assimilated	$\mu g\ Carbon$
dS	Change in satiation	h^{-1}
dz	Change in depth	m
F_{temp}	Carbon in pellet	$\mu g\ Carbon$
$\overline{I_g}$	Rate of Plankton ingestion	$plankton\ s^{-1}$
$I_{gmaxTotal}$	Combined ingestion rate	$plankton\ s^{-1}$
kd_{calc}	Used for daytime migration	Wm^{-2}
kn_{calc}	USed for nighttime migration	$none$
$Nitrogen_{ing}$	Nitrogen ingested	$\mu g\ Nitrogen$
$Nitrogen_{remin}$	Proportion of Nitrate to remineralise	$none$
$Nitrogen_{req}$	Proportion of Nitrate to assimilate	$none$
$OffspringCount$	Number of offspring	$none$
P_{can}	Probability of being cannibalised	$none$
$Reproduce$	Flag for reproduction	$none$
v_{mod}	Sign of satiation change	$none$
W_{TG}	Temperature and weighting factor	$none$
z_{day}	Depth change by daytime migration	m
z_{night}	Depth change by nighttime migration	m
z_{sink}	Depth change by sinking	m
z_{turb}	Depth change by turbulence	m

Figure 6.12: Local/Exported Variables for Copepods

6.5 Diatom Functional Group

Note that for all groups, rules are executed in each timestep, and are thus written with implicit units of 'per timestep'. To convert from hours to timesteps, the system parameter $\triangle t$ is used, which for the WB model is set to 0.5 hours.

6.5.1 Diatom Motion

Above the turbocline, diatoms are mixed by turbulence. We approximate turbulence by randomly locating the diatom between the surface and the turbocline. Below the turbocline, diatoms sink at a uniform rate. This behaviour occurs in all stages of diatom. Rules 6.2 shows the rules for the change in depth, during a timestep. Recall that all rules specify a change that happens in the course of one timestep, the length of which is defined by the variable $\triangle t$.

$$z \quad = \quad if\ (z \leq MLDepth)\ then\ z + z_{turb}\ else\ z + z_{sink} \tag{6.2}$$

The variable z is the depth of the particle, which is a state variable that all particles inherit. The identifiers, z_{turb} and z_{sink} represent the change in depth per timestep, due to turbulence, and sinking respectively. Here, we make use of sub-functions, creating one for turbulence, which calculates a value for z_{turb} and another sub-function for sinking, which calculates the value of z_{sink}, where the two variables are defined as *exported variables.*

Turbulence sub-function

$$z_{turb} = rnd(MLDepth) - z \tag{6.3}$$

The turbulence sub-function specifies the change in depth that happens to a diatom during a timestep due to turbulence. The approximation is that after a timestep, if the particle began above the turbocline, it will have a random depth between the turbocline and the surface. Hence z_{turb} is the difference between this final position, and its starting position, z.

Sinking sub-function

$$Z_{sink} = V_p \triangle t \tag{6.4}$$

Below the turbocline, diatoms sink at a steady rate, V_p, which is a constant parameter defined as 0.041667 mh^{-1}. As it is defined in units of absolute time, it must be multiplied by the size of the timestep.

6.5.2 Diatom Photoadaptation

$$I_m = I_m + \left(\frac{I_v - I_m}{t_a}\right) \triangle t \tag{6.5}$$

This rule models diatoms taking time to adapt to changes in light. I_m is a state variable for the diatom, encapsulating its current light adaptation level (Wm^{-2}). I_m asymptotically approaches the current ambient visible irradiance I_v, whether light or dark. The constant time parameter, t_a for diatom photoadaptation is 5 hours. The light adaptation variable, I_m is then used for calculating photosynthesis.

6.5.3 Diatom Energetics

The energy of a diatom increases by photosynthesis, but decreases by respiration, and cell division. As with motion, these three sub-processes are separated from the top-level energy rule.

$$Ec_{estimate} = Ec_{photo} - (Ec_{resp} + Ec_{cdiv}) \tag{6.6}$$

$$Ec_{max} = E_{max} - E_p \tag{6.7}$$

$$E_p = E_p + min(Ec_{estimate} \triangle t, Ec_{max}) \tag{6.8}$$

The first rule calculates an estimate of the new value for energy, using the three sub-functions. Ec_{photo}, Ec_{resp} and Ec_{cdiv} are the changes in energy due to the three energy processes, and hence are defined as *exported variables*, to be defined by the sub-functions. $Ec_{estimate}$ is defined here as a local variable, since it is only required within these three assignments.

Rule 6.7 calculates the maximum permitted change in energy, which is the difference between the current energy, the state variable E_p (J), and a parameter E_{max} (2.66x10^{-4} J), which defines the maximum energy.

Finally, the state variable for energy E_p is increased by the estimated change, or the maximum change if the estimate was too great.

Photosynthesis

$$Ec_{photo} = s_h \; k_F \; \pi \; {A_F}^2 \; I_V \; e^{-\frac{I_V}{I_m}} \tag{6.9}$$

Here, k_F is a light absorption constant, (0.63, no units), A_F is the radius of the diatom (1x10^{-5}m), I_V is the ambient irradiance in the visible spectrum (Wm^{-2}), and I_m is the light adaptation variable - the state variable defined in the light adaptation rule previously. π is a built in constant found under *System Constants* in figure A.6.

As ${A_F}^2$ is in m^2, and I_V is in Wm^{-2}, the right-hand side is measured in Watts. Ec_{photo} is converted from Js^{-1}into Jh^{-1} with the constant convertor s_h (3600 sh^{-1}).

Normal Respiration

In the winter months where the diatoms get less energy from photosynthesis, they reduce their respiration rate to save energy. This is modelled by a stage change, where the diatoms enter a *cyst* stage, in which a different respiration rule operates. Note that this is not in the original WB specification, and has been added after recent discussions [58].

$$Ec_{resp} = R_L \left(0.3 + 0.7\frac{T}{T_r}\right) \tag{6.10}$$

Rule 6.10 defines the standard respiration rule. R_L is a respiration parameter (2x10^{-7} Jh^{-1}), T is the ambient temperature and T_r is a reference temperature, $10°C$. The function is set to operate only in the *Living* diatom stage.

Cyst Respiration

The rule for respiration in winter uses a different respiration parameter R_C (1x10^{-7} Jh^{-1}), but is otherwise similar to standard respiration. It is set to operate only when

the diatom is in the winter (cyst) stage.

$$Ec_{resp} = R_C \left(0.3 + 0.7\frac{T}{T_r}\right) \tag{6.11}$$

Enter Cyst Stage

This function contains a single rule which causes the diatom to enter winter respiration mode between November and January of each year, inclusive. The variable d_{year} is a system variable representing the day of the year between 1 and 365; this is selected from the *system variables* section of figure A.6.

The parameter d_{cyst} is 31 days, representing the end of January, and d_{decyst} is 305 days, representing the start of November. If the day of the year is in the winter months between these two dates, then the diatom should change to the *Cyst* stage, for winter respiration. This function is set to operate only in the *Living* stage.

$$if\ (d_{year} > d_{cyst})\ or\ (d_{year} < d_{decyst})\ then\ change(Cyst) \tag{6.12}$$

Leave Cyst Stage

$$if\ (d_{year} > d_{decyst})\ and\ (d_{year} < d_{cyst})\ then\ change(Living) \tag{6.13}$$

In contrast to the *Become Cyst* function, the *Leave Cyst Stage* function contains a single rule which returns the diatom to the normal stage of respiration at midnight on the 1^{st} of February in each year of the simulation. This function is set to operate only in the *Cyst* stage of the diatom.

Note that the introduction of cyst behaviour into diatoms is a recent change, and has one flaw. The system variable d_{year} gives the number of days that have elapsed since the beginning of the current year, and d_{cyst} and d_{decyst} are numerically hard-wired to represent the beginning of November and the end of January respectively. The times at which diatoms enter and leave the cyst phase are specific to the climate, and the values of d_{cyst} and d_{decyst} assume that Winter occurs between November and January. This assumption is obviously not always the case, if the model is to be run in the Southern Hemisphere for example.

These values have been chosen to test the model with the Azores system, but ultimately a better way of establishing when to enter and leave the cyst phase will be required.

Cell Division

When diatoms perform cell division, a single diatom splits into two. There is an energy cost due to cell division, and the nutrients the original diatom contained are evenly divided between the divided cells.

Cell division is possible if the energy pool is above a certain threshold, and the amount of nitrogen absorbed by the diatom is above another threshold.

$$Nitrogen = A_{pool} + N_{pool} \tag{6.14}$$

The total absorbed nitrogen is calculated in rule 6.14; $Nitrogen$ is a local variable used just within this rule, but A_{pool} and N_{pool} are the pools automatically created within every particle when the ammonium and nitrate chemicals were introduced. See the rule for absorbing chemicals in section 6.5.4.

$$c_{div} = if\ (E_p > E_c)\ and\ (Nitrogen > N_c)\ then\ 1\ else\ 0 \tag{6.15}$$

Rule 6.15 decides whether cell division should occur, if the internal energy pool E_p has exceeded the necessary energy threshold E_c (1.4x10^{-4}J,) and if the nitrogen absorbed is greater than the threshold parameter N_c (4x10$^{-9}\mu$g of nitrogen). It remains at zero if cell division was not possible.

The variable c_{div} is defined here, and used three times in other rules. It is defined as a local variable, and like all variables in Planktonica, it is a floating point number (double). As its only values are 0 and 1, the option of providing boolean flags for such variables was considered. However, as this would require an extra variable type, and associated support for assignments and comparisons, we have chosen to keep all the data types numerical, and allow the user to define values for flags.

$$Ec_{div} = if\ (c_{div} = 1)\ then\ E_p - \frac{E_p - E_c}{2\ \triangle t}\ else\ 0 \tag{6.16}$$

Ec_{div} calculates the energy change if cell division has happened; this is for use in the energy rule 6.6. If cell division occurred, then the new value of E_p should be $\frac{E_p - E_c}{2}$. Dividing by $\triangle t$ yields the correct units for Ec_{div} (Jh^{-1}).

$$n_{red} = if\ (c_{div} = 1)\ then\ 0.5\ else\ 1 \tag{6.17}$$

$$N_{cdiv} = N_{pool}\ n_{red} \tag{6.18}$$

$$A_{cdiv} = A_{pool}\ n_{red} \tag{6.19}$$

Rule 6.17 defines a local variable n_{red}, which is used in rules 6.18 and 6.19 to halve the amount of nitrate and ammonium in the internal pools, should cell division happen. Here, N_{cdiv} refers to the amount of nitrate left, rather than the change in Nitrate, and is an exported variable, used in the rules for energy.

$$if\ (c_{div} = 1)\ then\ divide(2) \tag{6.20}$$

Finally, 6.20 calls the *divide* function, which performs the cell division, handling sub-populations behind the curtain, as described in section 3.4.2.

6.5.4 Diatom Nutrient Uptake

Diatom particles absorb both ammonium and nitrate together from the water. They have a maximum threshold, which the sum of their internal ammonium and nitrate must not exceed. Depletion handling rules performed behind the curtain prevent the diatom from absorbing more nutrient than was available, as discussed in section 3.3.2.

$$dN = \left(u_N \frac{N_{conc}}{N_{conc} + k_N}\right) \triangle t \tag{6.21}$$

$$dA = \left(u_A \frac{A_{conc}}{A_{conc} + k_A}\right) \triangle t \tag{6.22}$$

Rules 6.21 and 6.22 calculate the maximum amount of nitrate and ammonium respectively, that the diatom could absorb. The parameters u_n and u_a are maximum

uptake rates for nitrate and ammonium, both 4x10^{-10} μg Nitrogen h^{-1}, and the parameters k_N and k_A are half-saturation constants for nitrate and ammonium, both set to 0.5μ g Nitrogen m^{-3}. N_{conc} and A_{conc} are the chemical variables automatically created when the chemicals were created, and give the concentrations in each layer in μg Nitrogen m^{-3}. The results are stored in local variables dN and dA, which are local to this function.

$$dNitrogen = dA + dN \tag{6.23}$$

$$s = min\left(1, \frac{N_{pmax} - (A_{pool} + A_{uptake} + N_{pool} + N_{uptake})}{dNitrogen}\right) \tag{6.24}$$

The two uptakes are summed, storing the result, the total potential change in nitrogen, in the local variable dNitrogen. Rule 6.24 then calculates the difference between the maximum nutrient, N_{pmax} (7.6x10^{-9} μg Nitrogen), and the total nutrient in the diatom before this timestep, which is the sum of the pools, and the amounts gained from the previous timestep. The local variable s is the factor by which the nitrate and ammonium absorption must be reduced, if N_{pmax} would be exceeded.

$$if\ (dNitrogen > 0)\ then\ uptake(s\ dN, N) \tag{6.25}$$

$$if\ (dNitrogen > 0)\ then\ uptake(s\ dA, A) \tag{6.26}$$

Rules 6.25 and 6.26 model the absorbtion of nitrate and ammonium, via the *uptake* function, described in section 3.3.2.

$$N_{pool} = N_{cdiv} + N_{uptake} \tag{6.27}$$

$$A_{pool} = A_{cdiv} + A_{uptake} \tag{6.28}$$

Lastly, the nitrate and ammonium pools are updated in rules 6.27 and 6.28, using the *corrected* amounts gained from the previous timestep, and also the exported variables that carry the new value of the pool, including any changes caused by cell division (see section 6.5.3).

6.5.5 Diatom Chlorophyll Pool

This rule sets the concentration of chlorophyll within a single diatom to a constant value, k_{chl} (9.2x10$^{-10}\mu$ g Chlorophyll). Biofeedback uses this value as shown in section 3.3.1. Note that in the case of cell division, since the sub-population size of a particle doubles, but the chlorophyll pool defined per individual remains constant, it is assumed the cell dividing into two causes the amount of particulate chlorophyll in the system to double.

$$Chl_{Pool} = k_{chl} \tag{6.29}$$

6.5.6 Diatom Carbon Pool

The carbon content of a diatom is also assumed to be a constant, k_c (4.6x10$^{-4}\mu g\ Carbon$).

$$C_{Pool} = k_c \tag{6.30}$$

6.5.7 Diatom Mortality

Should the internal energy pool of the diatom drop below zero, the diatom dies. Here, the *change* function described in section 3.4.1 is used to change the classification of the diatom from the Living stage, to the Dead stage, which was setup in section 6.2. To prevent dead particles from contributing to biofeedback, the chlorophyll pool is set to zero on the diatom's death.

$$if\ (E_p < 0.0)\ then\ Chl_{pool} = 0.0 \tag{6.31}$$

$$if\ (E_p < 0.0)\ then\ change(Dead) \tag{6.32}$$

6.5.8 Diatom Remineralisation when dead

Bacteria release dead diatoms, causing their internal nutrients to be returned to the water. Since diatoms only absorb nitrate and ammonium, these are the only nutrients it is necessary to write remineralisation rules for.

$$release(A_{remin}\ A_{pool}\ \triangle t, A) \tag{6.33}$$

$$A_{pool} = A_{pool} - (A_{remin}\ A_{pool}\ \triangle t) \tag{6.34}$$

$$release(N_{remin}\ N_{pool}\ \triangle t, N) \tag{6.35}$$

$$N_{pool} = N_{pool} - (N_{remin}\ N_{pool}\ \triangle t) \tag{6.36}$$

A_{remin} and N_{remin} are the rates of remineralisation, both 0.00208333 μg Nitrogen h^{-1}, which are parameters defined by the user. The *release* function is used to write the changes to the chemical concentrations, since direct writing by the particles is forbidden (see section 3.3.2).

6.6 Copepod Functional Group

6.6.1 Copepod Motion

$$\begin{aligned} dz \;=\; & if\ (I_V > 0)\ and\ (z < MLDepth)\ then\ z_{turb} + z_{day}\ else\ (\\ & if\ (I_V > 0)\ and\ (z \geq MLDepth)\ then\ z_{sink} + z_{day}\ else\ (\\ & if\ (I_V < 0)\ and\ (z < MLDepth)\ then\ z_{turb}\ else \\ & z_{sink} + z_{night}\ )\ ) \end{aligned} \tag{6.37}$$

$$z = max(0, z + dz) \tag{6.38}$$

The motion of copepods is defined in terms of turbulence, sinking, migration by day, and migration by night, and as for diatoms, these auxiliary motion rules are delegated to subfunctions, with z_{turb}, z_{sink}, z_{day} and z_{night} defined as exported variables. If the particle is above the turbocline, then z_{turb} applies, otherwise z_{sink} is used, but note that copepods only sink when they are dead, as will be specified when making the sink rule. The value of z_{sink} will be zero while the copepod is in any of its live stages.

Day-time migration is used if the visible irradiance, I_V in the copepod's ambient environment is above zero, whereas if it is dark, night-time migration is applied, but only if the copepod is below the turbocline. Rule 6.38 then ensures that the copepod cannot escape from the top of the column, which can sometimes happen mathematically

(if not in nature), when turbulence and migration both cause upward motion of the copepod.

Day-Time Migration sub-function

During the day, copepods tend to stay deeper, in darker water where they are less visible to predators. However, if they are sufficiently hungry, they will take risks and venture into lighter water, where phytoplankton are more copious. The degree of satiation is represented by the copepod state variable S, which has no units, and ranges from zero for least satiated, to one for most satiated.

$$I_t = I_r(2 - S) \tag{6.39}$$

Rule 6.39 defines the target isolume; this is the level of light in Wm^{-2} that a copepod will aim to swim to, depending on how satiated it is. I_r (1 Wm^{-2}) is the default irradiance the copepods prefer when they are not hungry. Hence, as S varies between 0 and 1, the target isolume varies between I_r and $2I_r$. The hungrier it is, the more it will risk being seen by visual predators, in order to feed.

$$kd_{calc} = 0.4(I_V - I_t) \tag{6.40}$$

The copepod aims for this target isolume in a way represented by rule 6.40; I_V is the ambient visible irradiance (Wm^{-2}) and kd_{calc} is a local variable representing the difference between the current light, and the desired light.

$$\begin{aligned} z_{day} \quad = \quad & V_{max} \triangle t \ (if \ (kd_{calc} < -1) \ then \ -1 \ else \ (\\ & if \ (kd_{calc} > 1) then \ 1 \ else \ kd_{calc} \ ) \ ) \end{aligned} \tag{6.41}$$

The conditional part of rule calculates the sign of kd_{calc}; if the copepod is aiming to move up into lighter water, the sign is negative, whereas for moving downwards, the sign is positive. This is multiplied by a weighting factor, W_{TG} (see rule 6.6.2), which represents the effect of temperature and bodyweight on swimming velocity. As W_{TG} is

additionally used for night-time migration, and the ingestion rule, it is defined as an exported variable in a sub-function, described in section 6.6.2.

V_{max} is a parameter representing the maximum migration velocity, 45 mh^{-1}, and since the units are hour-based, $\triangle t$ is required.

Night-Time Migration sub-function

At night, copepods above the mixing layer are subject only to turbulence, but copepods below the mixing layer perform night-time migration. The variable v_{mod} is exported by the satiation subfunction defined in section 6.6.2, and represents whether the copepod is becoming hungrier, or less hungry over time; -1 implies the copepod is getting hungrier.

$$kn_{calc} = if\ (v_{mod} = -1)\ then\ -0.4(2 - S)\ else\ -1 \tag{6.42}$$

$$z_{night} = kn_{calc}\, W_{TG}\, V_{max}\, \triangle t \tag{6.43}$$

The local variable kn_{calc} represents how quickly the copepod would like to eat, and this is moderated by the maximum swimming speed V_{max}, and the effect of temperature and weight on the copepods swimming speed, W_{TG}, described in section 6.6.2.

Sinking sub-function

The sinking rule is similar to that of diatoms, and V_z has the same sinking value of 0.041667 mh^{-1}. The difference is that the sinking rule only applies to copepods that are dead, and also to copepod pellets.

$$z_{sink} = V_z\ \triangle t \tag{6.44}$$

Turbulence sub-function

The turbulence rule is identical to that of diatoms, and is applicable in all stages of copepod.

$$z_{turb} = rnd(MLDepth) - z \tag{6.45}$$

6.6.2 Copepod Ingestion

Refer to section 3.4.5 for information on the variable types necessary for ingestion. For the WB model, the ingestion requires the following steps.

The behaviour to model is that copepods ingest diatoms in their live and cyst stages, treating both stages similarly. The first step is to create a variety-based iterator, $\overline{P}^*$, which in VEW Species Builder (see section 7.3) will be set to contain live and cyst diatoms, of their default variety, as there is only one variety of diatoms and copepods in this version of the WB model. Recall that any rule in which $\overline{P}^*$ occurs, will be iterated for each element of $\overline{P}^*$, and when read, $\overline{P}^*$ returns the concentration of the member currently being considered.

$$\overline{I_g} = min \left(\frac{W_{TG} \overline{F} \int_{z_{[-1]}}^{z} \left(\begin{array}{ll} if \ \left(\overline{P}^* > \overline{P_{min}} \right) \ then & \frac{\left(\overline{P}^* - \overline{P_{min}} \right)^2}{\left(\overline{P}^* - \overline{P_{min}} \right) + \overline{k_I}} \\ else & 0 \end{array} \right) dz}{max \left(1x10^{-5}, \left| z - z_{[-1]} \right| \right)}, \overline{s_1} - \overline{s_2} \ S \right) \quad (6.46)$$

Since the right hand side contains $\overline{P}^*$ and other variety-based types associated with $\overline{P}^*$, the target variable, $\overline{I_g}$ must also be a variety-based variable associated with $\overline{P}^*$, as described in section 4.3.9. In this case, $\overline{I_g}$ is a variety-based state variable, and stores the ingestion rate of each plankton type in $\overline{P}^*$ (plankton s^{-1}).

This rule integrates a function between the starting and ending depth of the copepod's trajectory in the previous timestep; z is the current depth in metres, and $z_{[-1]}$ is the depth at the end of the *previous* timestep, obtained by a historic lookup (see section 4.3.10). Recall from section 3.4.4 that when the *integrate* function is used, $\overline{P}^*$ returns a concentration at the depth of the particle as it moves along its trajectory.

The ingestion rate is then averaged, by dividing by the change in depth, with a catch to prevent a divide-by-zero error in the unlikely case of the particle remaining at the same depth. The expression $(\overline{s_1} - \overline{s_2}S)$ gives the maximum ingestion rate, which any member of $\overline{I_g}$ cannot exceed. The parameters $\overline{s_1}$ and $\overline{s_2}$ (4.2 and 3.2 plankton s^{-1} respectively).

The ingestion rule uses a number of variety-based parameters, all of which are explicitly associated with $\overline{P}^*$ by the user. These are $\overline{F}$, the filtration rate (1x10^{-9}m^{3}s^{-1}), a half-saturation constant $\overline{k_I}$ (4x10^{6}plankton m^{-3}), the minimum concentration threshold $\overline{P_{min}}$ (1x10^{5}plankton m^{-3}) and W_{TG} is the weighting factor mentioned previously.

Having calculated the ingestion rate, the *ingest* function is used to carry out the reduction of other particles; refer to section 3.4.5.

$$ingest(\overline{P}^*, \begin{pmatrix} \overline{I_g} & if\ (\overline{P}^* > \overline{P_{min}}) \\ else & 0\ ) \end{pmatrix}) \tag{6.47}$$

The ingestion rates for all the target varieties, stored in I_{gv}, are summed into a single value, using the function *vSum*, one of the three variety-based reduction functions described in section 4.3.9.

Rule 6.47 is responsible for reducing the particles that have been ingested. This is done behind the curtain, reducing the sub-population sizes of the ingested prey, as described in section 3.4.5.

Satiation

Satiation defines a unit-less state variable S, representing how hungry a copepod is; 0 represents most hungry, 1 represents least hungry. This function begins with calculating the maximum ingestion rate. This is a variety-based calculation, as a copepod may have a different maximum ingestion rate of one particle-type to another. Firstly the maximum ingestion rate for both live and cyst diatoms is set; I_{gmax} is associated with $\overline{P}^*$.

$$\overline{I_{gmax}} = \overline{s_1} - \overline{s_2}S \tag{6.48}$$

$I_{gmaxTotal}$, in rule 6.49 is a state variable for copepods, representing the total number of individuals per second that could have been ingested by the copepod. It is calculated using the special *vSum* function, which takes as its argument a variety-based variable. In this case, I_{gmax} is set of maximum ingestion rates for each particle concentration in

the set $\overline{P}^*$.

$$I_{gmaxTotal} = vSum\left(\overline{I_{gmax}}\right) \tag{6.49}$$

Rule 6.50 defines the change in satiation. The carbon content per cell, k_c is assumed constant as ($4.6\text{x}10^{-4}\mu g$ Carbon per plankton), for both live and cyst diatoms. The parameter t_m is a relaxation time for copepod ingestion (4 hours).

$$\begin{aligned} dS \quad = \quad & if\ (I_{gmaxTotal} > 0)\ then\ \frac{\left(\frac{C_{ingest}}{I_{gmaxTotal}\ k_c\ s_h\ \triangle t} - S\right)}{t_m} \\ & else\ \frac{1-S}{t_m} \end{aligned} \tag{6.50}$$

The result of rule 6.50 is stored in a local variable dS, so it can be used in both rules 6.51 and 6.52. Rule 6.51 simply applies the change to the satiation state variable.

$$S = S + dS\triangle t \tag{6.51}$$

Finally, v_{mod} is an exported variable representing whether the copepod is getting more or less hungry; it is unitless and set to 1.0 or -1.0. Recall this was used for copepod migration; see section 6.6.1. Hence, satiation is made a sub-function so that v_{mod} is available for other functions.

$$v_{mod} = if\ (dS < S_{min})\ then\ -1\ else\ 1 \tag{6.52}$$

Weighting Function

$$W_{TG} = 0.3 + 0.7\frac{T}{T_r}\left(if\ (C_{pool} > G_{max})\ then\ 1\ else\ \left(\frac{C_{pool}}{G_{max}}\right)^{0.7}\right) \tag{6.53}$$

The weighting function produces a unitless exported variable, W_{TG} which represents the effect of water temperature and weight on the swimming speed of a copepod, (see sections 6.6.1) and its ingestion, (see section 6.6.2). It is calculated using the copepod's carbon pool, C_{pool} (μg Carbon), its maximum weight G_{max} (100μg), the ambient temperature T and a reference temperature T_r ($10°C$).

6.6.3 Copepod Respiration

$$R = R_{ass} + R_{Bas} + R_{Bac} \tag{6.54}$$

Copepod Respiration is separated into three parts, which are summed in rule 6.54 to give a state variable R, the carbon lost by respiration per hour (μgCarbon h^{-1}). Firstly, there is a respiration cost of assimilating (ingesting), R_{ass}. This is calculated using the amount of carbon ingested in the previous timestep, converted into the amount of carbon ingested per hour, by dividing by $\triangle t$. Two assimilation parameters k_a (1, no units), and R_s (0.3, no units) are also used.

$$R_{ass} = R_s \; k_a \; \frac{C_{ingest}}{\triangle t} \tag{6.55}$$

Secondly, R_{bas} is the basal respiration rate, calculated using a basal respiration parameter R_b ($0.3\text{x}10^{-3}$ h^{-1}), the weighting function W_{TG}, and the maximum weight of the copepod, G_{max} (100μg Carbon).

$$R_{Bas} = R_b \; W_{TG} \; {G_{max}}^{0.7} \tag{6.56}$$

Finally, R_{bac} is the background rate of respiration, which uses a background respiration parameter k_b (0.1, no units). The three respiration rates are all local variables, only used within this function.

$$R_{Bac} = R_b \; k_b \; {G_{max}}^{0.7} \tag{6.57}$$

6.6.4 Copepod Growth and Cannibalism

Here, the weight change of copepods is handled, and if ingestion of diatoms is not sufficient to overcome respiration loss, then the copepods cannibalise each other. Firstly, the amount of nitrogen ingested is calculated by summing the amounts of nitrogen in the ammonium and nitrate that have been ingested. All units here are in μg Nitrogen. $Nitrogen_{ing}$ is a local variable used only within this function, whereas A_{ingest} and N_{ingest} are the automatically added ingestion variables for ammonium and nitrate.

$$Nitrogen_{ing} = A_{ingest} + N_{ingest} \tag{6.58}$$

Next, the carbon assimilated this timestep, C_{Ass} is calculated. It uses the assimilation parameter k_a (1, no units), the amount of carbon ingested in the last timestep, C_{ingest}, and the carbon loss due to respiration R (μg Carbon h^{-1}) converted into per-timestep units by multiplying by the timestep, $\triangle t$.

$$C_{Ass} = k_a \left(C_{ingest} - R \triangle t\right) \tag{6.59}$$

Not all the nitrogen gained by ingesting diatoms can be assimilated. The local variable $Nitrogen_{req}$ is the proportion of nitrogen to be assimilated (no units). If any nitrogen was ingested, then the amount assimilated is proportional to the amount of carbon assimilated (C_{Ass}). The ratio r_{ac} is a parameter to convert carbon into nitrogen (8.7x10^{-6} μg Nitrogen μg Carbon^{-1}).

$$Nitrogen_{req} = \ if\ (Nitrogen_{ing} > 0)\ then\ \frac{r_{ac} max\left(C_{Ass}, 0\right)}{Nitrogen_{ing}}\ else\ 0 \tag{6.60}$$

The proportion of the ingested nitrogen to be remineralised is then calculated, using the carbon losses due to background and basal respiration, (R_{Bac} and R_{Bas}), converting carbon to nitrogen again with the parameter r_{ac}.

$$Nitrogen_{remin} = \ if\ (C_{Ass} > 0)\ then\ \frac{\left(R_{Bac} + R_{Bas}\right) r_{ac}}{Nitrogen_{ing}}\ else\ 0 \tag{6.61}$$

If the respiration losses were greater than the carbon assimilated by ingestion, then copepods eat each other to make up the deficit. If this is the case, the probability of an individual copepod being eaten is P_{can} (unitless), defined below.

$$P_{can} = \ if\ (C_{Ass} < 0)\ then\ -\frac{C_{Ass}}{C_{pool}}\ else\ 0 \tag{6.62}$$

The carbon, nitrate and ammonium pools are now increased by the amounts of each chemical assimilated.

$$C_{pool} = \ if\ (Reproduce = 1) \quad then \quad C_{pool} = G_{max} \tag{6.63}$$
$$else \quad C_{pool} + max(C_{Ass}, 0)$$

$$N_{pool} = N_{pool} + Nitrogen_{req} N_{ingest} \tag{6.64}$$

$$A_{pool} = \begin{array}{lll} if\ (Reproduce = 1) & then & A_{pool} - G_{min} r_{ac} \\ & else & A_{pool} + Nitrogen_{req} A_{ingest} \end{array} \tag{6.65}$$

The proportion of ingested nitrogen to be remineralised is then released to the ambient environment as ammonium.

$$release(Nitrogen_{remin} A_{ingest}, A) \tag{6.66}$$

$$release(Nitrogen_{remin} N_{ingest}, A) \tag{6.67}$$

Finally, the probability of cannibalism is implemented using the *pchange* function, causing the proportion P_{can} of the sub-population to move into the Cannibalised stage.

$$if\ (C_{Ass} < 0)\ then\ pchange(Cannibalised, P_{can}) \tag{6.68}$$

6.6.5 Copepod Excretion

The carbon that wasn't assimilated is excreted as a faecal pellet. The amount of carbon, F_{Temp} (μg Carbon) is calculated below.

$$F_{Temp} = (1 - k_a\,(1 - R_s))\,C_{ingest} \tag{6.69}$$

If there is a positive amount of carbon, a pellet is created. The pellet's ammonium pool is calculated using the amount of carbon to be excreted, and the conversion parameter r_{ac}.

$$\begin{array}{ll} if\ (F_{Temp} > 0)\ then & create(Pellet, 1) \\ & A_{pool} = r_{ac}\ F_{Temp} \end{array} \tag{6.70}$$

6.6.6 Copepod Maturity

Copepods move from the *juvenile* to the *adult* stage where their carbon pool, C_{pool} (μ g Carbon) reaches a threshold weight G_{max} (100μg Carbon). This rule changes their stage, and is called only when the copepod is in the *juvenile* stage.

$$if\ (C_{pool} > G_{max})\ then\ change(Adult) \tag{6.71}$$

6.6.7 Copepod Aging

$$A_r = A_r + \triangle t \tag{6.72}$$

When copepods become adults, a biological clock starts counting, and reproduction and senility happen at set times. A_r counts the number of hours that have elapsed since the copepod became an adult, and continues counting until the copepod dies. It is thus called in the adult and senile stages only.

6.6.8 Copepod Reproduction

Having reached the adult stage, a gestation time begins, after which eggs hatch (reproduction); this gestation time is defined by the parameter A_I (480 hours). A numerical flag is used to record whether reproduction should occur, and is defined as an exported function, so that it can be used in the *growth* function, hence Copepod reproduction is defined as a sub-function.

$$Reproduce = if\ (A_r = A_I)\ then\ 1\ else\ 0 \tag{6.73}$$

Upon reproduction, a number of offspring are produced, depending on the carbon gained by the copepod since gestation began, and the minimum carbon for the newborn copepods, G_{min} (0.2 $\mu g\ Carbon$).

$$OffspringCount = if\ (Reproduce = 0)\ then\ \frac{C_{pool} - G_{max}}{G_{min}}\ else\ 0 \tag{6.74}$$

$$\begin{aligned} if\ (Reproduce = 1)\ then \quad & create(Newborn, OffSpringCount) \\ & S = 0.5 \\ & A_r = 0 \\ & C_{pool} = G_{min} \\ & R = 0 \end{aligned} \tag{6.75}$$

The offspring are created in the *newborn* stage, and the satiation (S), age (A_r), ingestion rate (I_g), respiration loss (R) and carbon pool(C_{pool}) of the newborn copepods are initialised.

6.6.9 Copepod Newborn Mortality

When copepods are born via the reproduction rule, they start in the stage called *Newborn*. The user sets this mortality rule to only apply to the newborn stage. The copepods stay *newborn* only momentarily; during this stage, a proportion I_{mp} (0.7) of the newborn copepods die due to infant mortality. The remainder progress immediately to the juvenile stage.

$$pchange(Dead,\ I_{mp}) \tag{6.76}$$

$$change(Juvenile) \tag{6.77}$$

6.6.10 Copepod Senility

When the time since a copepod became mature reaches A_I (480 hours), it enters the senile stage. This function is only called when copepods are in the adult stage.

$$if\ (A_r > A_I)\ then\ change(Senile) \tag{6.78}$$

6.6.11 Death by Senility

When copepods are senile, they die linearly over time. This rule is defined as a probability of dying that increases over time, such that 480 hours after becoming an adult, a copepod has will have a death probability of 1.

$$pchange(Dead, \frac{1}{L - A_r}) \tag{6.79}$$

The first time this rule is called, A_r is 480 hours - the value of A_I. (See section 6.6.10). L is a constant parameter, set at 960 hours, so the initial probability of death is $\frac{1}{480}$. After another 479 hours, A_r will be 959 hours, and the probability of death will be 1.

6.6.12 Death by Cannibalism

When copepods are cannibalised, the contents of their pools are assimilated by the cannibal in rules 6.64 and 6.66. Their pools must be set to zero, and they enter the *dead*

stage. Differently to senility or infant mortality, their pools are not remineralised, but are fully assimilated by the cannibal.

$$A_{pool} = 0 \tag{6.80}$$

$$N_{pool} = 0 \tag{6.81}$$

$$C_{pool} = 0 \tag{6.82}$$

$$change(Dead) \tag{6.83}$$

6.6.13 Dead Copepod Remineralisation

$$release(A_{pool}, A) \tag{6.84}$$

$$A_{pool} = 0 \tag{6.85}$$

$$release(N_{pool}, N) \tag{6.86}$$

$$N_{pool} = 0 \tag{6.87}$$

When copepods die, they are assumed to immediately remineralise their pools into their ambient environment. Death occurs either by infant mortality, or senility.

6.6.14 Pellet Remineralisation

$$release(A_{remin}\ A_{pool}\ \triangle t, A_{conc}) \tag{6.88}$$

$$A_{pool} = A_{pool} - A_{remin}\ A_{pool}\ \triangle t \tag{6.89}$$

$$release(N_{remin}\ N_{pool}\ \triangle t, N_{conc}) \tag{6.90}$$

$$N_{pool} = N_{pool} - N_{remin}\ N_{pool}\ \triangle t \tag{6.91}$$

Here, the ammonium and nitrate pools are remineralised at rates A_{remin} and N_{remin}, both 0.00208333 μg Nitrogen h^{-1}. For each, the internal pool is reduced, and the same amount is remineralised into the ambient environment using the *release* function (See section 3.3.2). As both dead copepods and pellets have their pools remineralised by bacteria in this way, the function is set to run in both the *Dead* and *Pellet* stages.

6.7 Top Predators

This section describes the way the issue of trophic closure is addressed by Planktonica. The method of modelling closure has been recently formulated by John Woods [58]. The extension has been made to the original WB model in collaboration with Matteo Sinerchia and Adrian Rogers. The support for top predators is included in Planktonica, but the interfaces within VEW Scenario do not yet support closure. Hence, this section should be considered as contribution towards future work.

Representing trophic closure requires some extensions to the scenario, in which the existence of the predators (size, concentration and location) is defined. The biological model then defines the behaviour of the top predators - ingestion and remineralisation.

6.7.1 Method

A single Lagrangian-Ensemble particle is created in each biological layer, representing a sub-population of top predators in that layer. The size of the sub-population needs to be set by the user, which is not possible using conventional planktonica rules, since sub-population sizes can only be changed indirectly using the API. The predators have ingestion rules similar to those for copepods ingesting diatoms.

This requirement is met by making the size, concentration, and position of top predators part of the scenario. From the VEW Scenario application, rules for these three functions can be written, which cause the creation of three variables; the size S_t, the total number of predators, N_t, and a function that defines the vertical distribution of top predators, D_t.

In VEW Designer, any functional group can be defined to be a top predator group, by use of a tickbox. At this point, the size variable S_t becomes available for use in rules, and the ingestion rules for the top predators are written in the usual way. D_t and N_t however are not exposed to the user; they are used internally. This is described below, alongside the rules for trophic closure in the new WB model.

6.7.2 Modifications to the Scenario

Size Function

The size of the top predators for WB is defined as follows:-

$$S_t = if\ (d_0 < d_{year})\ and\ (d_{year} < d_{max}) \quad then \quad s_0(p+1)^{d_{year}-d_0} \qquad (6.92)$$
$$else \quad 0$$

As mentioned, it depends on the day of the year, d_{year}, which counts the number of days that have elapsed since January 1^{st} of the current year, hence S_t is part of the scenario. The predators exist for a season of the year between day d_0 (121 days), and d_{max} (221 days). The initial size is defined by parameter s_0 (3 mm), and p is the bodyweight percentage growth (0.07, no units). Hence, the size of the predators increases over time.

Total Concentration

A single rule is used to define the total concentration of predators, N_t. The predators first become 'active' at d_0, and they decrease exponentially with e-folding time, d_{star} set to 5, until they have all died off at day d_{max}.

$$N_t = N_{back} + \quad if\ (d_0 \leq d_{year})\ and\ (d_{year} < d_{max}) \qquad (6.93)$$
$$then\ \ (n_0 - N_{back})\, e^{-\frac{(d_{year}-d_0)}{d_{star}}}\ \ else\ 0$$

The value of N_t is then used to position the predators at the beginning of each timestep, using the vertical distribution function below.

Vertical Distribution

The vertical distribution defines the proportion of the total concentration assigned to each depth.

$$D_t = if\ (z < 100)\ then\ 0.01\ else\ 0 \qquad (6.94)$$

Behind the curtain, this rule is executed at the beginning of the timestep in the following way. For each predator in existence (which is one per biological layer), its sub-population size c is set to the value of $D_t N_t$, where z is the depth of the top predator being considered, and N_t is the total concentration previously defined. Since there is one top predator per layer, and layers are one metre tall in the WB model, the total D_t of all the top predators is 1, and the total of all the sub-populations will be equal to N_t; this must be the case.

6.7.3 Modifications to the Biological Model

Ingestion

The ingestion rules for top predators are below. The weighting function is similar to that we have seen previously for copepods, but the weight ratio used previously is replaced with a function of the size of the predators, S_t, and the maximum size, S_{max} (40 mm). T refers to the ambient temperature, and T_r is a reference temperature ($10°C$).

$$W_{TG} = (0.3 + 0.7(\frac{T}{T_r}) + \quad (if\ (S_t > S_{max}0\ then\ 1 \quad else\ \left(\frac{0.225(0.000194 * S_t^{2.59})}{S_{max}}\right)^{0.7} \tag{6.95}$$

The variety iterator, $\overline{P}^*$ here contains three members: juvenile, adult and senile copepods. $\overline{P_{min}}$ is the minimum concentration threshold for ingestion ($plankton\ m^{-3}$), which here is set to zero for all three stages of copepod to be ingested. The half-saturation constant k_I is set to 4x10^{-6} plankton m^{-3}.

$$\overline{I_g} = min \quad (W_{TG} 1x10^{-5} \left(\begin{array}{lll} if\ \left(\overline{P}^* > \overline{P_{min}}\right)\ then & \frac{\left(\overline{P}^* - \overline{P_{min}}\right)^2}{(\overline{P}^* - \overline{P_{min}}) + \overline{k_I}} \\ else & 0 \end{array} \right), \tag{6.96}$$

$$S_t 0.6156 e^{-0.0321(d_{year} - d_0)})$$

$$ingest(\overline{P}^*, \left(\begin{array}{ll} if\ (\overline{P}^* > \overline{P_{min}}) & \overline{I_g} \\ else & 0 \end{array} \right)) \tag{6.97}$$

Remineralisation

As the top predators assimilate the contents of the copepods they ingest, these need to be remineralised to the ambient environment in each layer. The carbon, nitrate and ammonium that were ingested, are all remineralised as ammonium, using the conversion factor r_{ac}.

$$release(C_{ingest}r_{ac} + N_{ingest} + A_{ingest}, A) \tag{6.98}$$

6.8 Particle Management and Scenario Settings

The model is completed by adding particle management and scenario settings into other interfaces described in chapter 7. These settings are as follows.

- Diatom particles are set to have an initial sub-population size of 25000, distributed at 20 particles per metre between 0 and 200m, and they begin in their live stage. This is a total of 4000 particles.
- Particle management splits both living and cyst diatoms when 20 or fewer particles (of that stage) exist in a layer, and merges whenever more than 40 particles exist in a layer.
- Dead diatoms are merged so that there is at most 1 in a layer.
- Copepods are initialised with a sub-population size of 17, distributed between 0 and 200m at 3 particles per metre; a total of 600 particles. They begin the simulation as juveniles.
- For newborn, juvenile, adult and senile particles, the particle management is set to merge whenever there are more than 600 particles of that stage in the column. However, this condition is never met in practice, since the time between reproduction and death by senility is always less than the time between birth and reproduction.

- Copepods in the dead, pellet and cannibalised stage however are merged down to 1 particle per layer, since they simply sink and remineralise.
- Top predator initialisation is not yet supported by VEW Scenario, so for the present, the initialisation has been manually coded into the XML file; Planktonica compiles this successfully. Top predators are created at 1 particle per metre, with sub-population size defined by the rules in section 6.7.
- Particle management is set to merge predators when there is more than one per layer, but this will not be necessary in practice since the top predators do not move.
- The location of the simulation used for experiments in this thesis is the Azores ecosystem at 41°N, 27°W, and the water column is fixed at this point.
- Simulations begin at 0600 hours, on March 1^{st} 1995.

Chapter 7

The Virtual Ecology Workbench

The modelling language described in the previous chapter is only of practical value when it is compiled into an executable simulation, which will require initial conditions and climate data, and if the results can be analysed to show meaningful emergent properties. The purpose of this chapter is to describe how this is done by the other components of the framework that Planktonica fits into: the Virtual Ecology Workbench (VEW).

The VEW is a suite of applications that together allow design and execution of plankton ecosystem simulations and analysis of the results. In this chapter, the composition of the VEW is explained. This thesis contributes the modelling language (the engine of the VEW), the model builder, the compiler and an interactive debugging tool, but for complete design and analysis of simulations, other applications are required, and have been written by a team of other authors. Table 7.1 shows the set of applications, their purposes and authors.

The specification for a model is stored in a single XML document. This document is created by VEW Designer, and the other applications in the VEW may extend the XML. When all the applications have added their sections, compilation can be performed, in which Java source files and binary data files are created. The Java source files are then compiled into classes, and are packaged together with the binary data files into a single JAR file. The JAR file then contains the entire specification, input data and execution code for a model instance.

Component	Purpose	Author
Modelling Language	The rule-building language for models	Wes Hinsley
VEW Controller	Project management, and guides model building	Adrian Rogers
VEW Designer	The interface for building models	Wes Hinsley
VEW Species Builder	Specifies parameterisations of functional groups	Adrian Rogers
VEW Particle Manager	Specifies particle management rules	Adrian Rogers
VEW Scenario	Climate Data (Boundary Conditions), Events	Adrian Rogers
VEW Data Viewer	Climate Data Visualiser	Matteo Sinerchia
VEW Output Control	Output options, logging	Adrian Rogers
VEW Compiler	Compilation of complete model	Wes Hinsley
VEW Run Control	Launching batches of simulations	Adrian Rogers
VEW LiveSim	Interactive visualisation of models	Wes Hinsley
VEW Analyser	Off-line visualisation of results	Adrian Rogers
VEW Documenter	Produces PDF specification of model	Evan Weaver

Figure 7.1: The components of the VEW

7.1 VEW Controller

VEW Controller is the first interface met by a VEW user, shown in figure 7.2. It allows the creation of a new model, perhaps based on a previously created model, or the alteration of models that have not yet been completed. Both new creations and alterations are handled by launching VEW Designer. Once models have been completed, and projects have been created using that model, then VEW Controller prevents further alteration of the model, thus ensuring consistency between the results obtained from running a model, and the source model.

After a model has been created using VEW Designer, VEW controller allows any number of projects to be made using that model; a project exists purely to provide

organisation, allowing the user to give a name to a certain investigation being carried out with a certain model. Within each project, any number of named experiments may be created, where an experiment defines an instance of a simulation: the parameterisation of the model to be used.

VEW Controller then guides the user sequentially through creating parameterisations of functional groups, initialisation and particle management, defining the scenario for the model, the logging options, and finally compilation and execution.

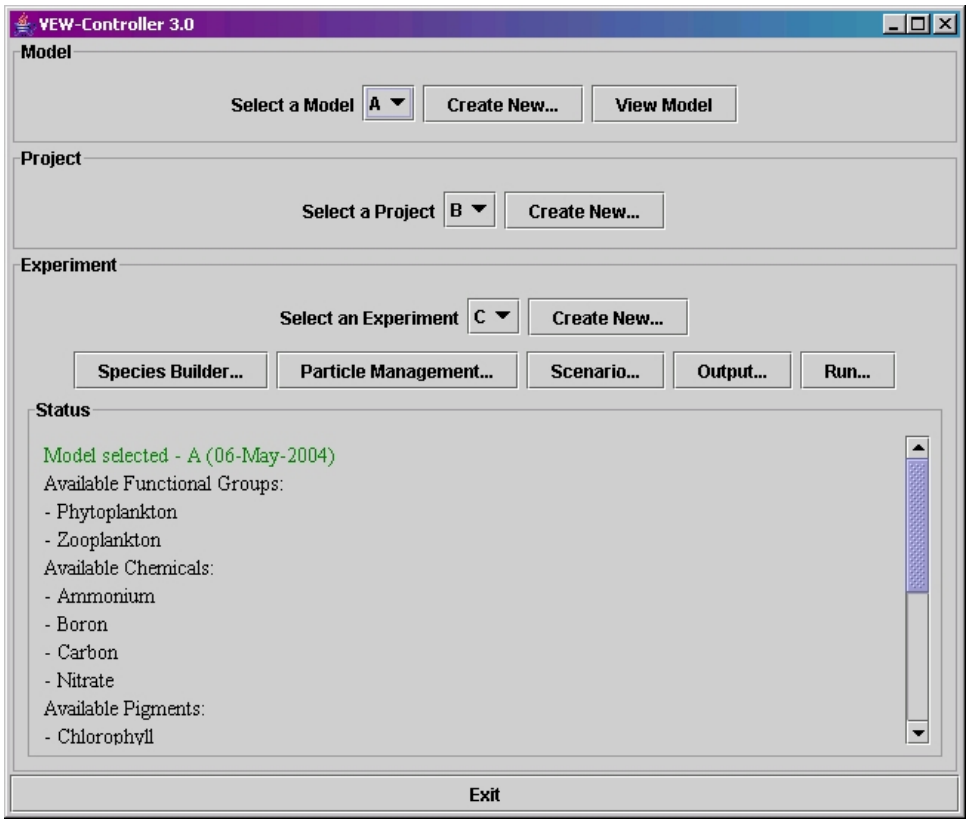

Figure 7.2: VEW Controller

7.2 VEW Designer

VEW Designer provides the interfaces for creating functional groups, chemicals (possibly with action spectra), and the rules, separated into functions and sub-functions. New variables or parameters may be created, and VEW Designer requires a default value and units to be specified.

An important property of VEW Designer is that it is customised to the job of creating Planktonica rules, rather than being akin to a generic equation editor. It prevents the creation of invalid rules by only allowing the user to choose from a menu of valid options, at each stage of rule creation. For example, when defining an assignment, VEW Designer will only allow the user to choose a variable on the left hand side, and a numerical expression on the right hand side. Alternatively, when defining a conditional functional, VEW Designer only allows a boolean expression on the left hand side, and a function on the right hand side. We believe this makes model building easier for the user.

Figure 7.3 shows the basic composition of the XML document after the user has created the model using VEW Designer; a single kernel tag, and a number of functional group tags and chemical tags exist.

7.3 VEW Species Builder

After using the model builder to specify the functional groups, chemicals and pigments in the system, *VEW Controller*, initiated by Arun Rishi [42], and largely developed by Adrian Rogers [44], is then used to specify species of the functional group, and varieties of each species, which have been described in figure 3.8.

For each species, the values of a and b for each parameter are set, and for each variety, the value of the base parameter, x, is set. At least one species and variety must exist for all functional groups; these are created by default. Any number of additional species and varieties may then be created.

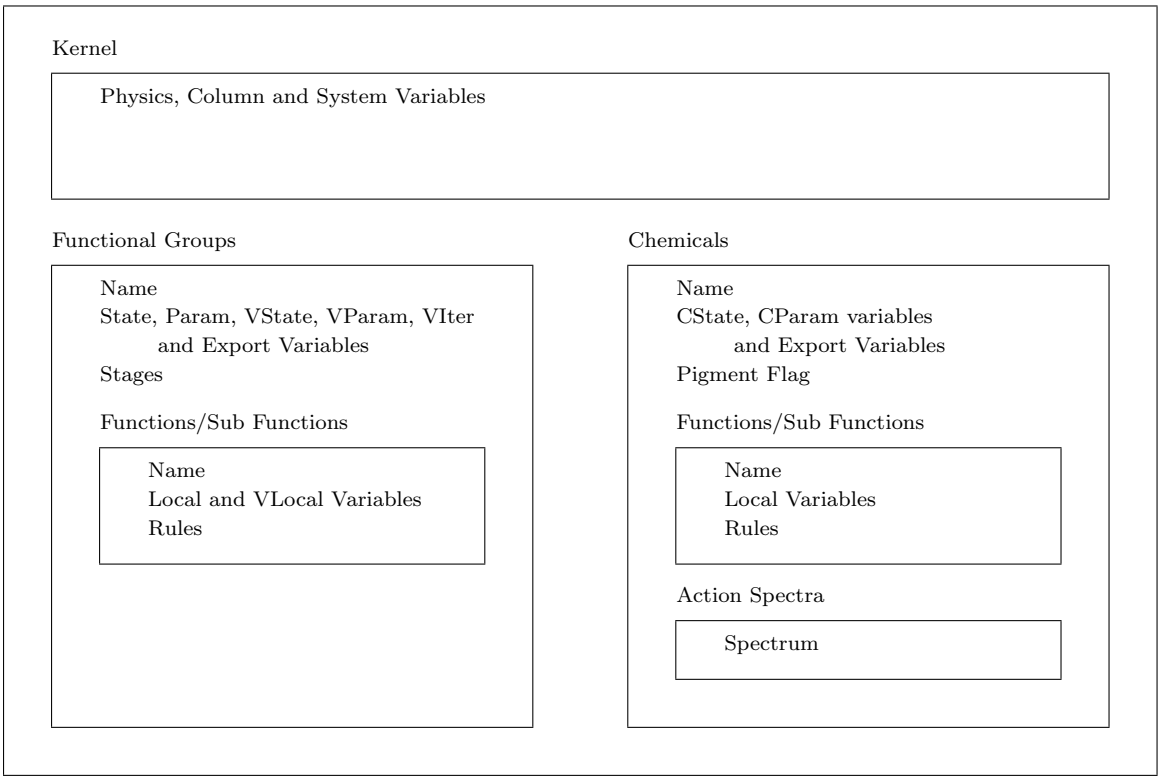

Figure 7.3: XML Document after VEW Designer

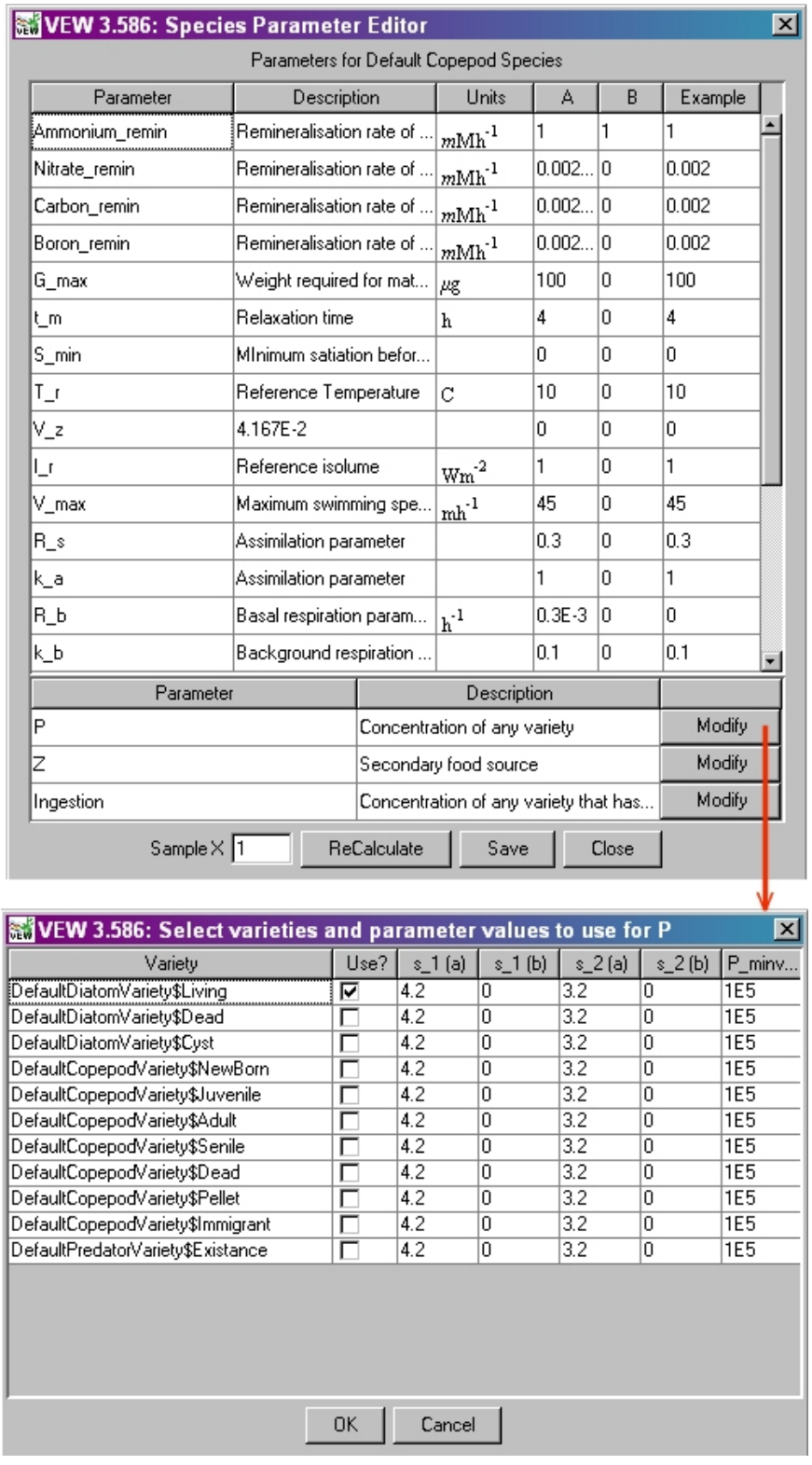

Figure 7.4: Setting Variety Iterators and Variety-Specific Parameters in VEW Species Builder

Figure 7.4 shows the way that the values for parameters are set. If variety-based variables have been used, then the variety iterator set, described in section 4.3.9, is shown in the parameters screen, where each stage of each variety can be included among the varieties that the iterator set refers to. The values for each variety-dependent parameter may then be set as shown.

The changes that the Species Builder introduces to the XML Document are shown in figure 7.5, specifically a number of species tags may be added within each functional group, and a number of variety tags may exist within the species tag.

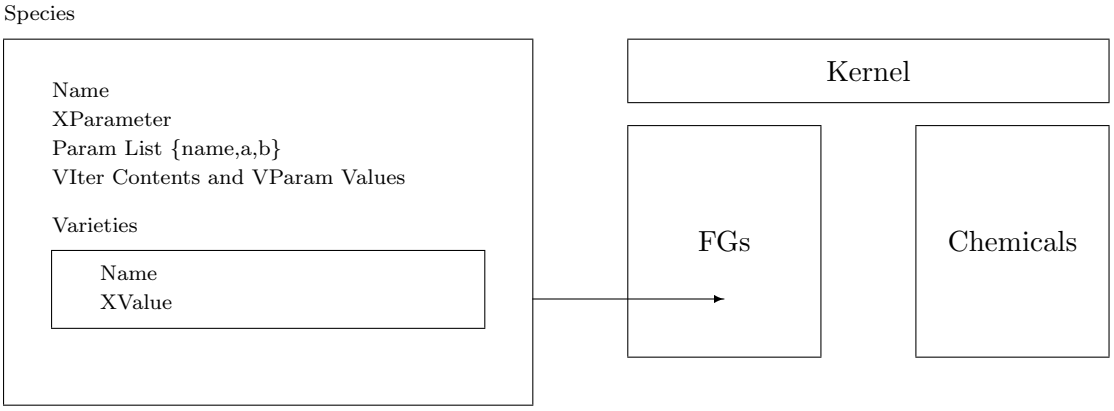

Figure 7.5: Changes made to XML Document by VEW Species Builder

7.4 VEW Particle Manager

Here, the *initial conditions* and *particle management* conditions for each variety may be set up. Figure 7.6 shows where the initial concentration of particles per metre is set, (where each particle is a sub-population), and the depth boundaries between which the particles are initially distributed. Also the starting size of the sub-population, and values for all of the state variables are set here. The default values are those the user gave when creating the variables in VEW Designer. In the case of variety-based state variables, separate entries appear for each stage of each variety. If the upper and lower limit values are different, then the random number generator (see section 4.5.5) is used to pick a value between the limits.

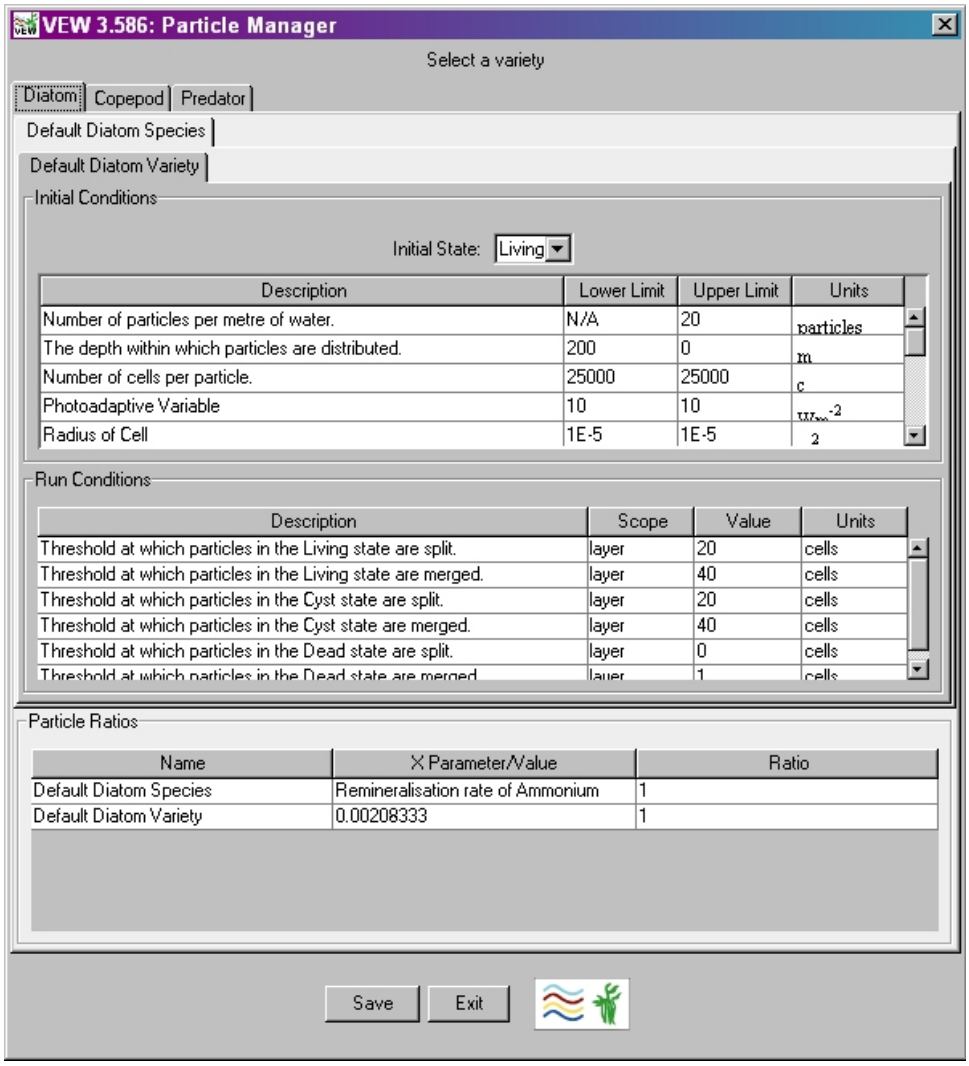

Figure 7.6: Particle Management and Initialisation

The *run conditions* define particle management rules during the simulation. These rules allow variation of the number of individual trajectories, thus affecting demographic noise, while avoiding excessive computational cost.

Two methods are provided; the first splits the largest sub-populations should the concentration fall below a threshold, and the second merges the smallest sub-populations should the concentration rise above a threshold. The thresholds are set by the user in

VEW Controller, and the scope of particle management can be to each layer individually, or to the whole water column as if it were a single layer.

Various sorting algorithms are used to implement this efficiently, and other research projects [33] have investigated alternative methods of controlling demographic noise.

The XML document is changed as shown in figure 7.7; for each variety, the initial particle stage, and the general particle management rules are added. General rules will exist for each stage of the functional group.

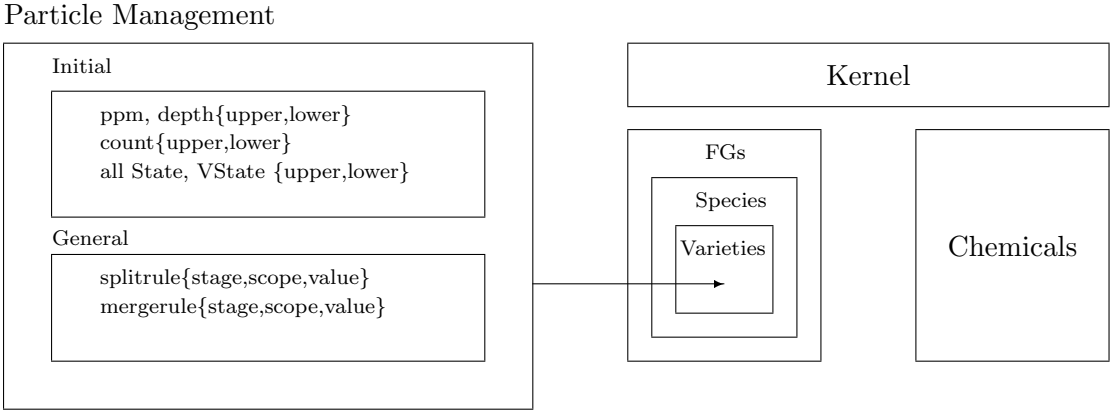

Figure 7.7: Changes made to XML Document by VEW Particle Manager

7.5 VEW Scenario

The VEW Scenario component shown in figure 7.8 is now used to define parameters for the water column itself. The main map shows the physical position of the water column in the ocean, which here has been set to a fixed location using the tracking mode menu on the right. Alternatively, a forward integration can be set, where the column starts at the specified position, and moves around the ocean, or even a backward integration where a starting position is calculated such that the column ends at the specified position at the time specified at the bottom. These settings define the input data used for calculating the physical properties of the column, as described in section 3.3.1.

The scenario is the specification of the *external* conditions for the model; it is un-

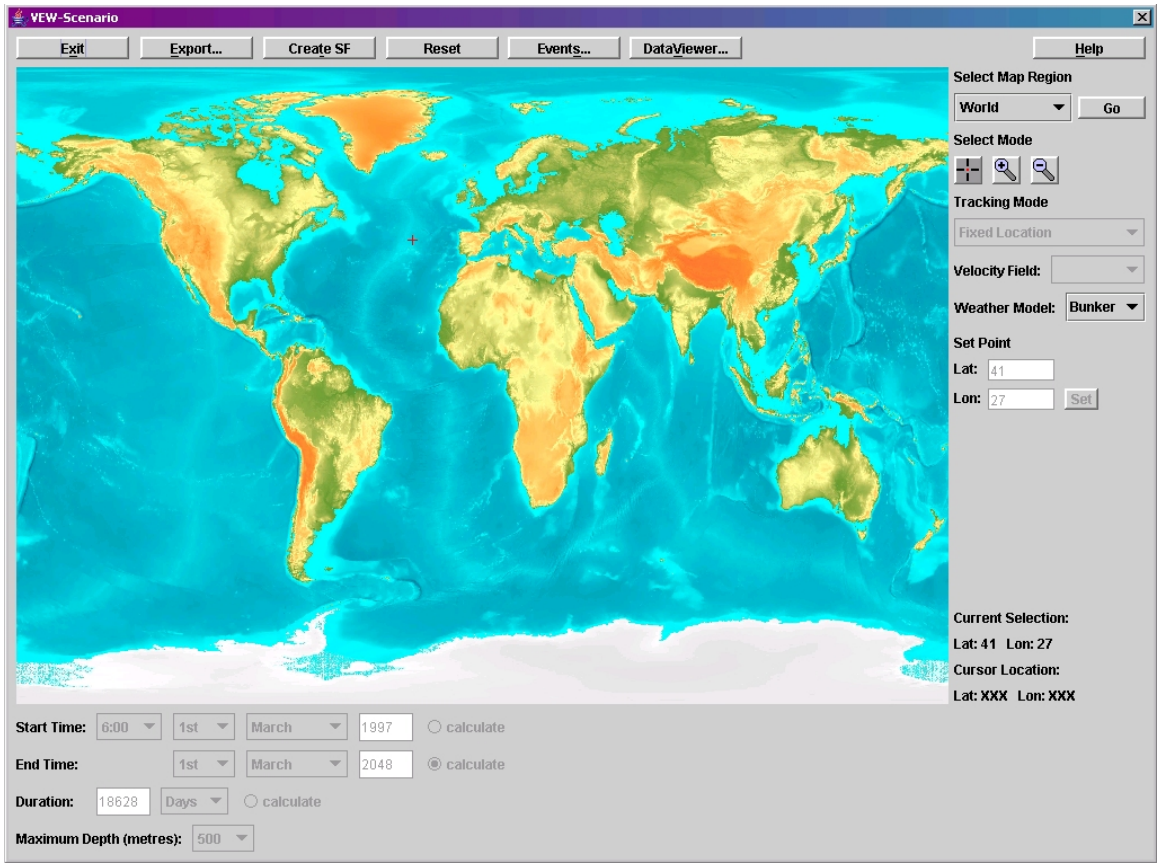

Figure 7.8: VEW Scenario

affected by feedback from the ecosystem. It consists of two parts; firstly the *initial conditions* describe the complete initial state of the simulation. This includes the time, and associated setup of the physical properties of the column, the chemical concentrations and the initial placement of different particle types.

The *boundary conditions* define the forcing climatological data which the simulation reads each timestep. From this data, along with biofeedback, the physical properties of the column are calculated. The VEW Scenario application provides these conditions as binary files, produced when the user specifies the location and path of the water column for the duration of the simulation.

To simulate any kind of external input to the water column, such as one-off oil spills

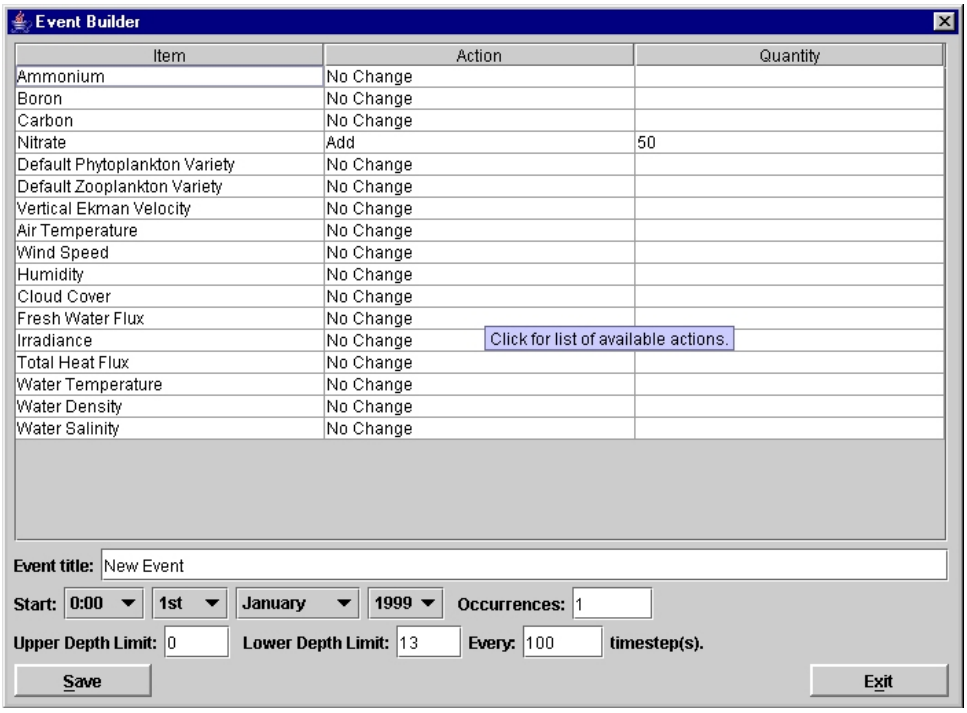

Figure 7.9: VEW Scenario Event Manager

or continual regular doses of pollution, events are provided. An event is any forced change to a property in the simulation, such as an increase in a certain chemical, or an increase in the number of a certain variety of plankton. One special event exists at the beginning of all simulations; it contains definitions for the initial chemical concentrations (or chemical profiles) in each layer.

Figure 7.9 shows the event manager interface. Events can be set to affect a certain proportion of the water column (upper and lower depth limit). They can start at any time and date in the simulation, repeating with a certain frequency, until a certain number of occurrences has been reached. An event with zero occurrences causes the events to continue forever at the specified frequency.

The scenario and events interfaces add a new block to the XML document, as shown in figure 7.10. The information added includes everything necessary to recreate the scenario from nothing; the time and duration data, location, what type of trajectory

the column should take, along with the chemical profiles and any other events the user created.

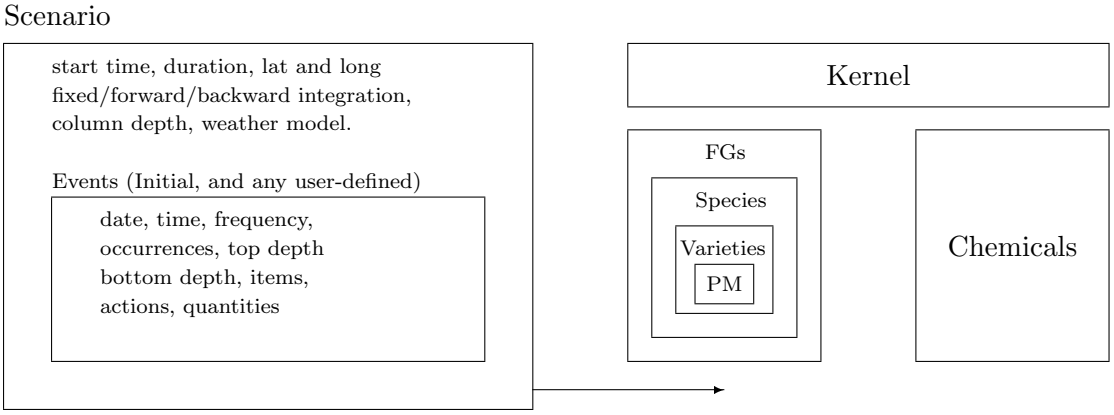

Figure 7.10: Changes made to XML Document by VEW Scenario and Event Manager

7.6 VEW Data Viewer

The Data Viewer, figure 7.11, made by Matteo Sinerchia [46] lets the user view the data that Scenario will use as input data. This includes visualisation of the nutrient profiles, velocity fields, and Levitus profiles for temperature. The current data set that can be visualised is the climate data of the world between 1961 and 2001. This is purely a visualisation tool for Scenario; it makes no changes to the XML document.

7.7 VEW Output Control

Any variable in a VEW simulation can be logged for analysis; depending on the number of particles in the simulation and the number of internal state variables that are required for analysis, the output data could become very large, and carry a resulting performance cost. For the WB model, a simulation with full output data produces about a gigabyte of data per year.

The VEW Output Control interface, shown in figure 7.12, allows the user to define

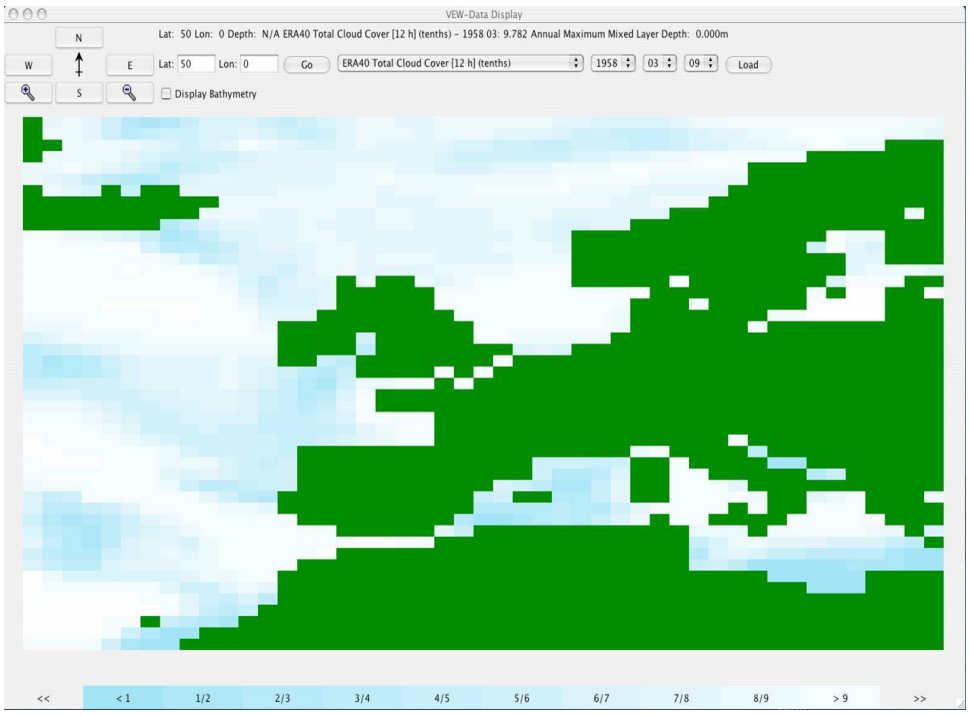

Figure 7.11: VEW Data Viewer

which variables are required for analysis. Figure 7.13 shows the changes made to the XML document. Separate logging options are set for every variety. For each, the range of variables that can be logged is separated into the state variables of functional groups, the ambient physical properties, ambient concentrations of varieties, and ambient chemical concentrations; recall that ambient means at a particle's instantaneous depth. The 'environment' group logs the values of physical, chemical or biological properties for the whole column, allowing profiles of the column to be plotted.

In either case (variety or environment group), for each group of variables to log, the user gives a start and end time, between which logging of that group is active, and the number of timesteps between samples (called frequency). Finally, a list of variables within that group is included, giving the name, description, and a flag to set whether logging is enabled on that variable.

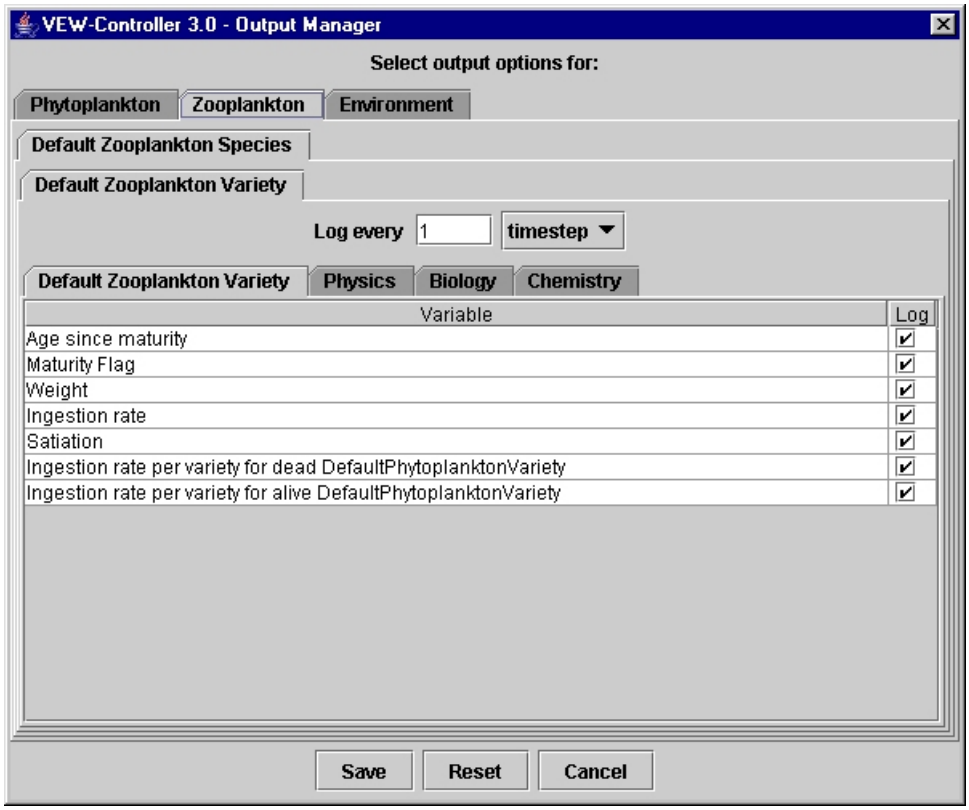

Figure 7.12: Output Options

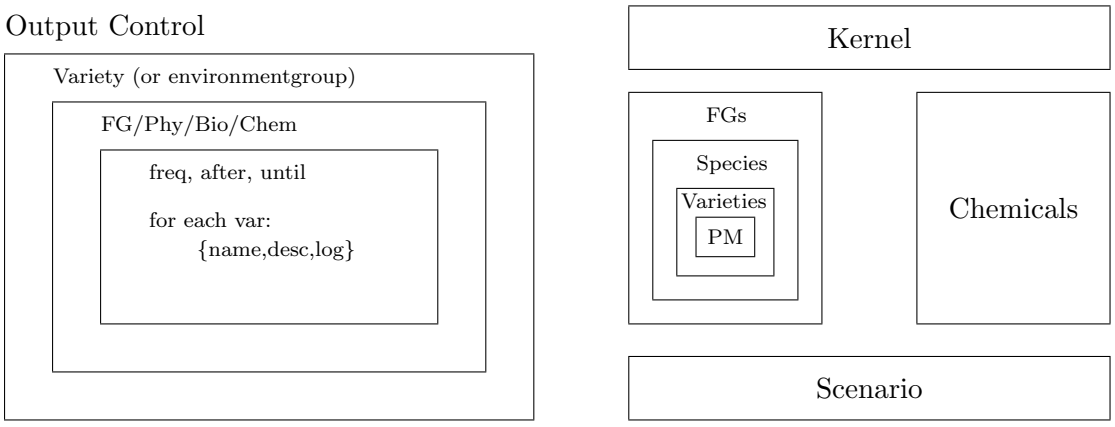

Figure 7.13: Changes made to XML Document by VEW Output Control

7.8 VEW Compiler

Planktonica's compiler is a contribution of this thesis. It performs a single pass of the XML document, producing a set of new classes for functional groups, species, varieties, their variables and parameters, chemicals and pigments, and a single system class used for initialisation, scenario and logging options. These new classes, along with a set of predefined classes for the physics, data structures, and file handling, are compiled with the standard *javac* compiler to produce an executable code.

The classes are then placed into a single JAR file, along with the binary data produced by Scenario, and a copy of the model XML file, by the VEW Controller application, when VEW Compiler finishes.

7.9 VEW Run Control

For executing the compiled model code, the VEW Run Control interface is used. At present, the VEW does not generate code that can be run in parallel, however for experiments concerning noise and stability, it is often necessary to run the same experiment many times with different random seeds. Similarly, many useful investigations can be made by running models that are identical except for the value of one parameter.

The run controller facilitates this, allowing the user to spawn simulations on various target machines, with different random seeds or parameter values.

7.10 VEW LiveSim

LiveSim is a contribution of this thesis: a front-end to drive the simulation. At any time, the user can retrieve virtually all of the data regarding the current state of the system. This data includes so far:-

- Internal state variables for all particles in simulation, including sorting particles by any variable.
- Column properties (sunlight, zenith height, turbocline, etc.)

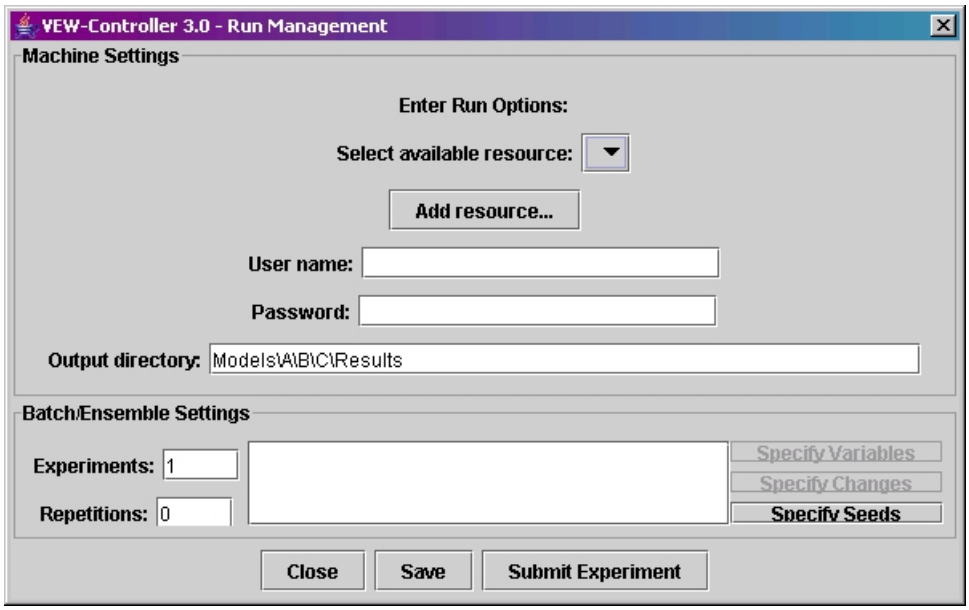

Figure 7.14: Run Manager

- Column total populations for functional groups, species and varieties in each stage.
- Physical properties over depth (temperature, density, etc.)
- Chemical and pigment concentrations over depth in particle's internal pool, or in solution.
- Concentrations of different particle types over depth, in each stage.
- Histograms for sub-population size, to check for particle management anomalies.

.

This list has been extended over the course of debugging, so that it already contains many useful diagnostics for debugging models. LiveSim then lets the user step through a complete timestep, or run until a certain time, or step more slowly through the different routines that occur during a timestep, to see the changes as they occur. The screenshot in figure 7.15 shows the state of the top-most live diatom in the main window, and the profile of copepod concentration with the water column surface on the left, depth increasing to the right. The red line shows the turbocline depth.

LiveSim has been extremely useful, if not crucial for rapid debugging of models. Although it has been designed as a developer tool, it has recently been redeveloped as an M.Eng Computing group project, with particular focus on the user interface and the range of different graphs that can be plotted over time [37].

7.11 VEW Analyser

VEW Analyser, originally developed by Sarah Bennett as VEWData, and more recently re-written by Adrian Rogers [44], allows a large range of standard plots to be made. Analyser has been tailored over the years to provide the graphs that are most useful to the research needs met so far. It differs to LiveSim in that it is executed after a simulation has finished, and produces a much wider set of diagnostic graphs.

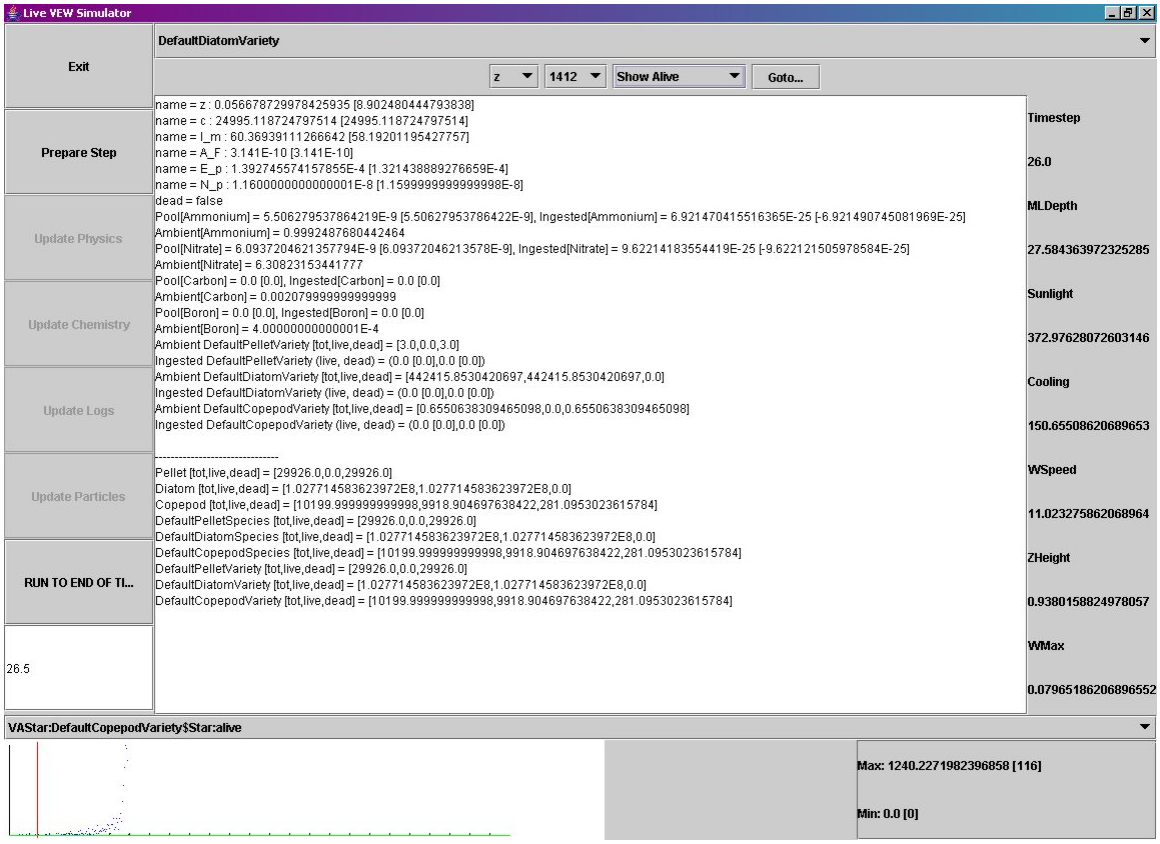

Figure 7.15: LiveSim 26 hours into the WB Model

While some planning was required to ensure that the simulation produces output in the correct format, Analyser itself is not a contribution of this thesis. A screenshot is shown in figure 7.16, and for further information refer to other documentation [43].

7.12 VEW Documenter

This utility was conceived early in this work, involving converting the model specification file into a Latex document, so that having made a model, a PDF file would automatically be created, listing the rules, and the variable types used in each rule, along with their descriptions and units.

That utility was substantially extended recently by Evan Weaver to include preview

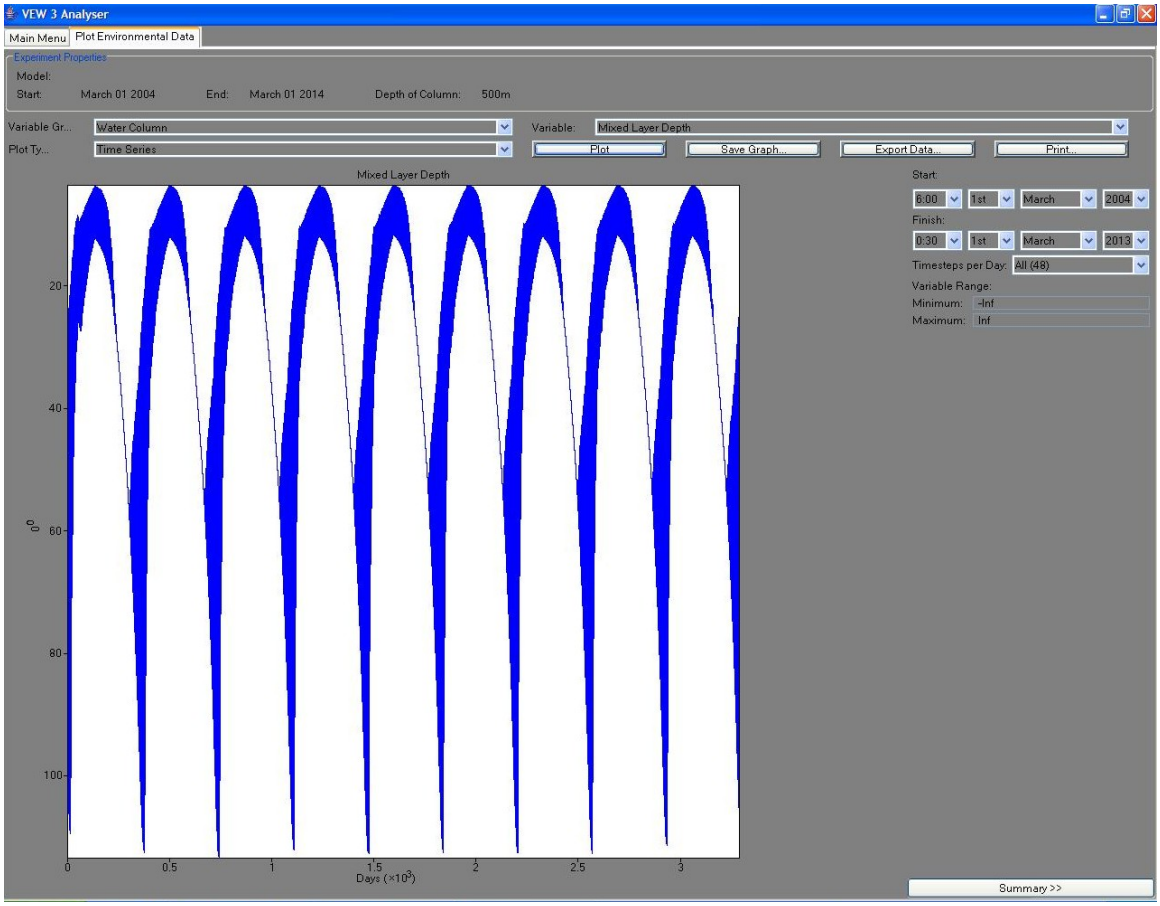

Figure 7.16: VEW Analyser

graphs of each rule, and diagrams of the whole model, rather than just the particle rules.

To support Evan's work, an extra run-mode was added to the simulation, that if activated, keeps track of maximum and minimum run-time values of all variables; this was necessary for plotting graphs at the correct scale in the output document.

Chapter 8

Evaluation

This chapter presents some results from experiments created using Planktonica, and the VEW. The results of running the implementation of WB described in chapter 6 are described first. The purpose is to demonstrate that the particles in the simulation behave as their rules suggest.

The following two sections describe how models can be built onto the WB model; the first experiment provides a very crude modelling of an oil spillage, and the second adds a delay between ingestion and excretion in the copepods. The purpose of these two sections is to demonstrate the ease with which new experiments can be made, and to outline how the process of model development, integration and analysis may be carried out using the VEW.

8.1 Results of new WB Model

In this section, a selection of results from the new WB model (chapter 6) are presented, beginning with environmental data for solar irradiance and ammonium concentration. Following that are audit trails showing the behaviour of a single diatom or copepod in its environment. Finally, six years of data from the diatom and copepod yearly cycle are plotted.

The solar irradiance for a year in the simulation is shown in figure 8.1. The graph

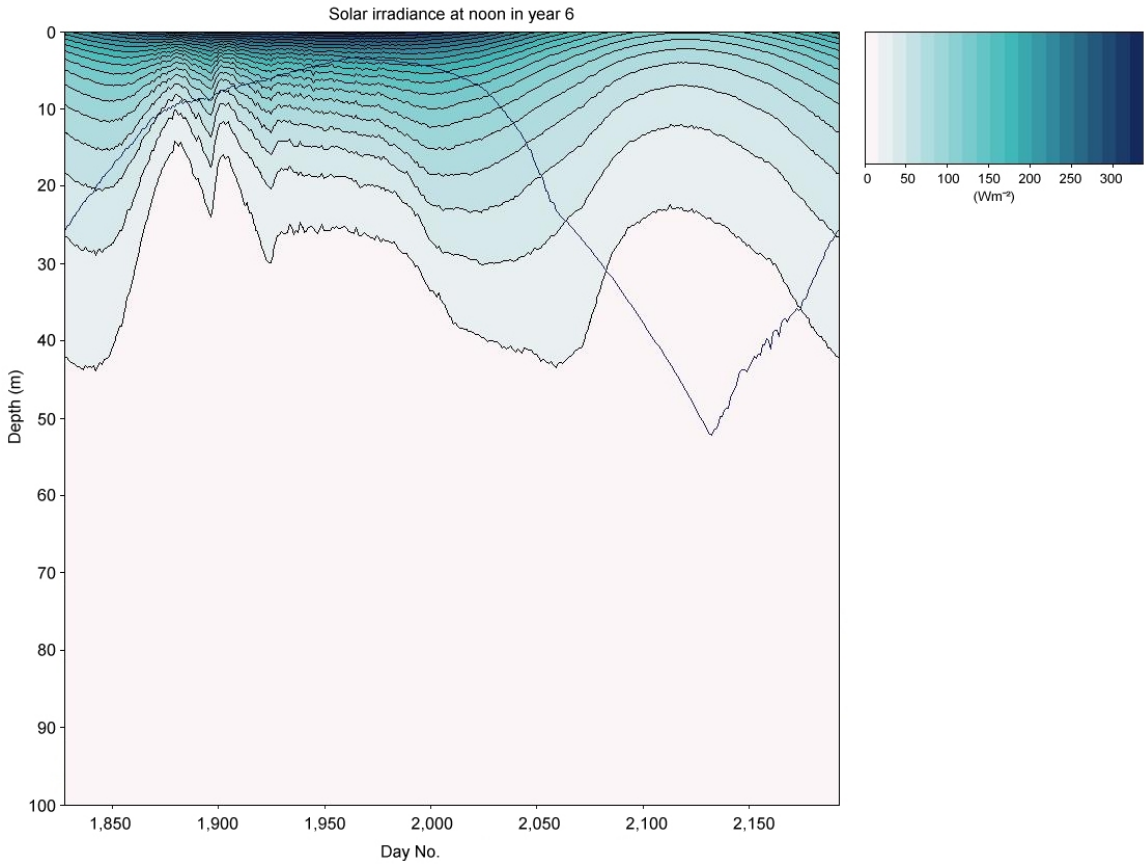

Figure 8.1: Visible irradiance at noon during year 6, starting on March 1st

is taken from the 6th year of a simulation (March 1^{st} 2000 to March 1^{st} 2001) with readings taken at noon each day.

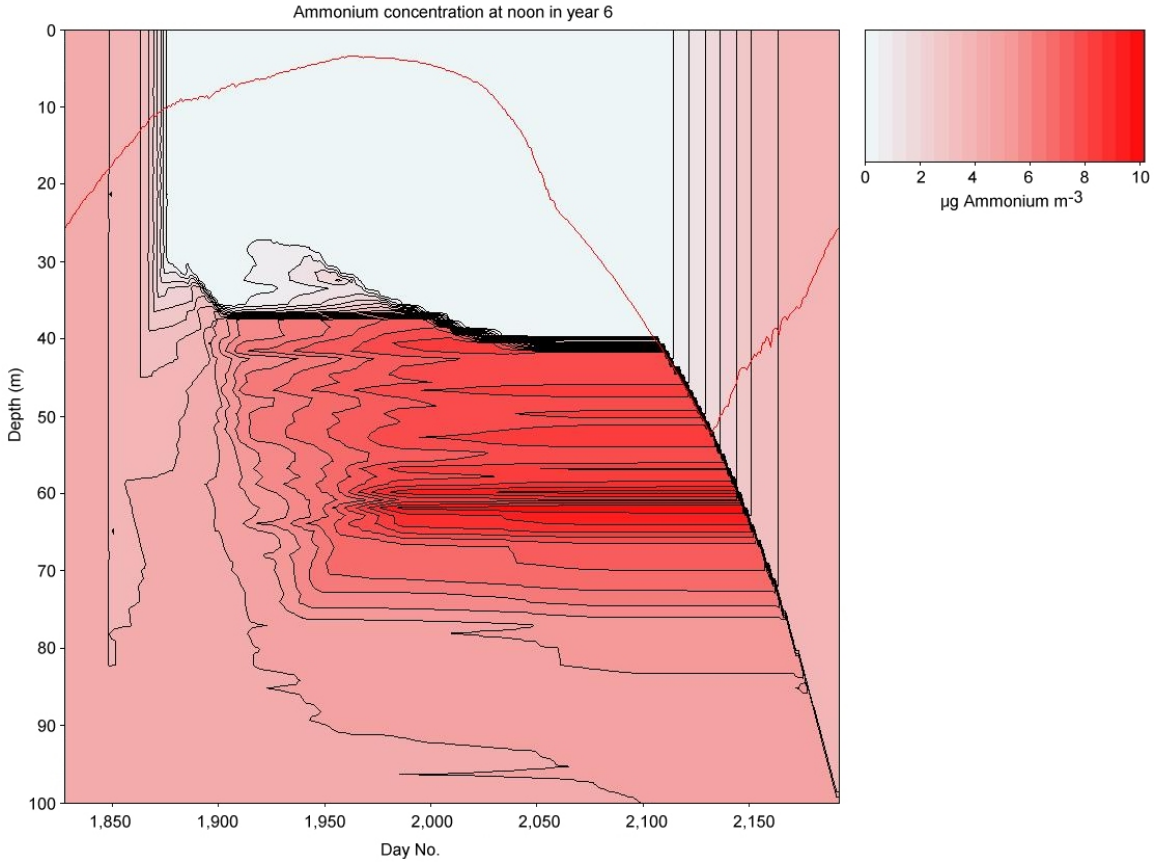

Figure 8.2: Concentration of Ammonium at noon during year 6, starting on March 1st

The ammonium concentration in the top 100 metres in the 6th year is shown in figure 8.2. The diatom bloom depletes the ammonium in the top thirty metres, and it is only later returned as the turbocline drops, and ammonium that was remineralised or excreted by copepods is returned to the mixing layer.

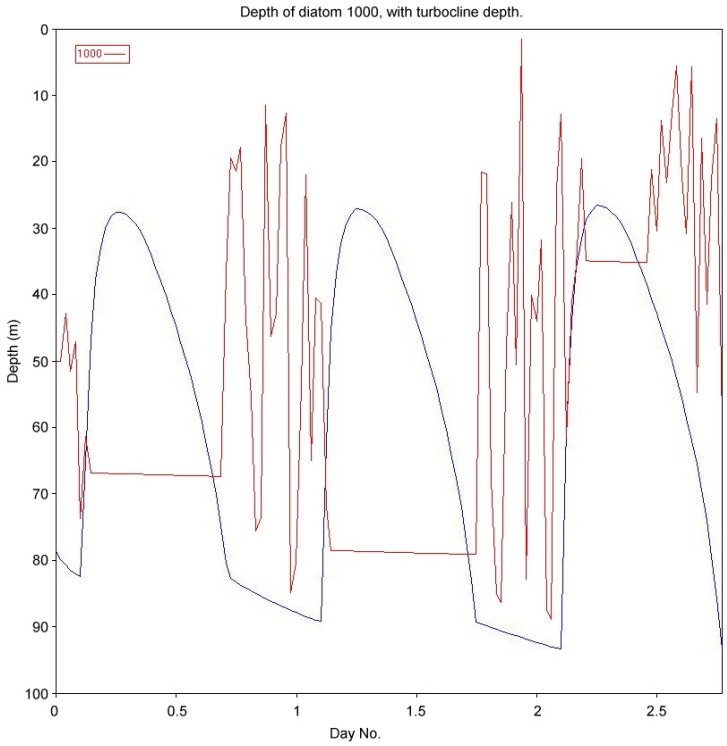

Figure 8.3: Audit trail for depth of a diatom, with turbocline

Figure 8.3 shows the depth of a single diatom, as described in rule 6.2. The diatom is randomly displaced when above the turbocline (rule 6.3), and sinks uniformly below the turbocline (rule 6.4). The slight delay between the turbocline falling and the turbulence affecting the particle is caused by the history; the turbulence rule uses the value of the turbocline from the previous timestep.

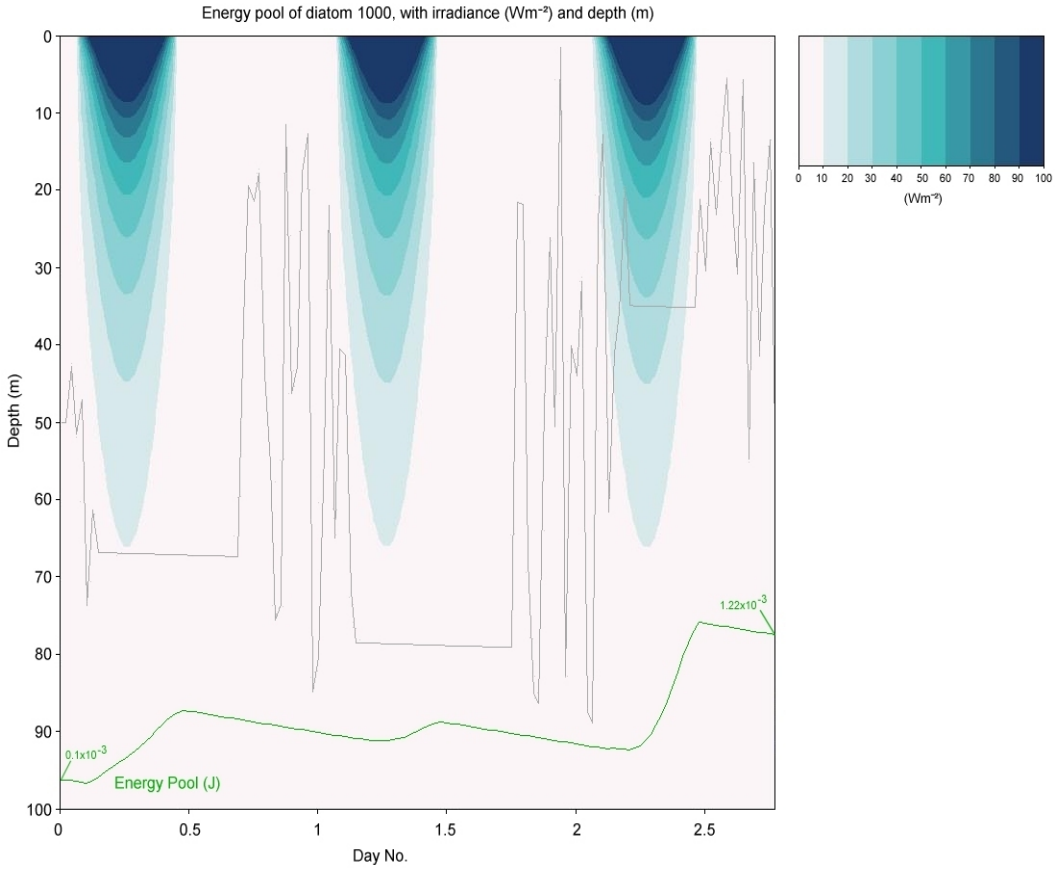

Figure 8.4: Audit trail for energy of a diatom, with irradiance and depth

In figure 8.4, the energy of a diatom is plotted (see section 6.5.3), along with its depth superimposed on the solar irradiance. The closer the depth is to the brighter light, the steeper the rise in energy is, due to photosynthesis (rule 6.9). When the particle is further away from the light, energy losses due to respiration (rule 6.10) take over and the energy level drops.

The energy curve lags a little behind the changes in irradiance; this is due to light adaptation (rule 6.5), by which the diatom takes a short period of time to respond to changes in irradiance.

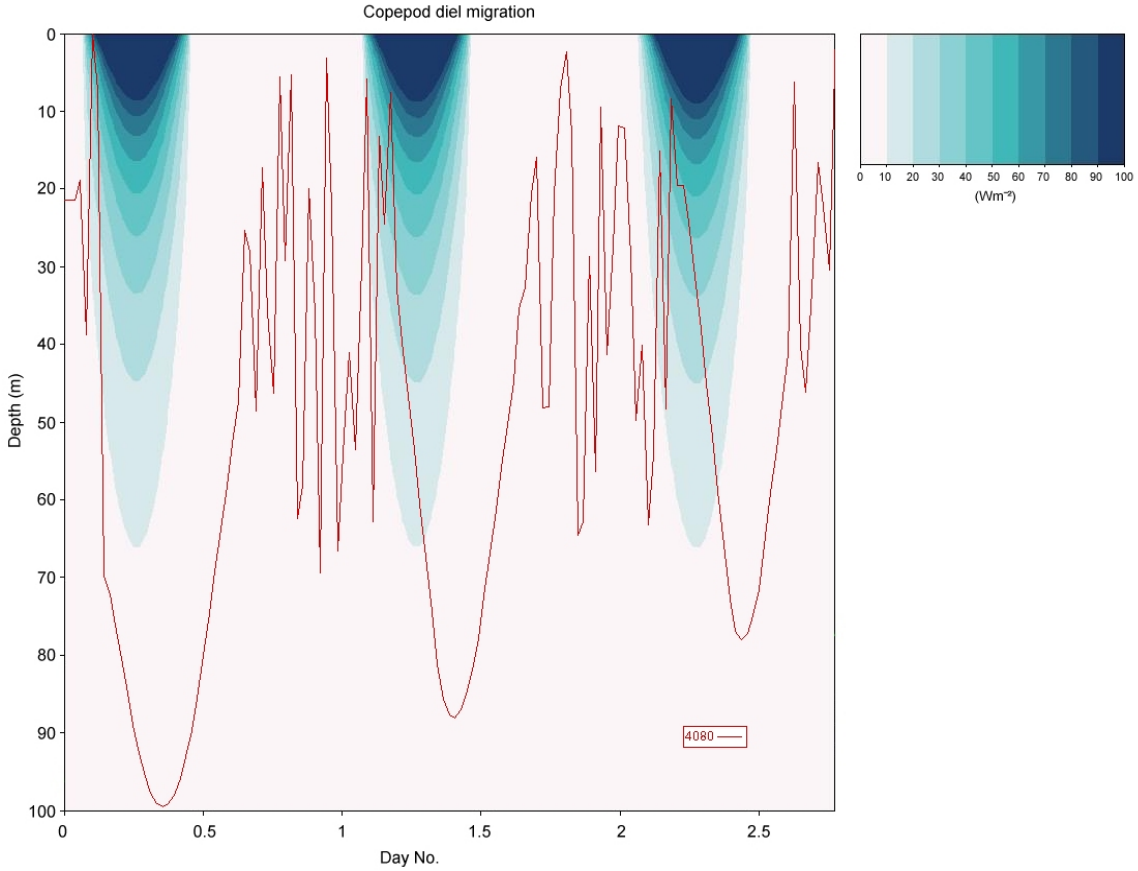

Figure 8.5: Diel migration of a copepod

Figure 8.5 shows a copepod performing diel migration, as in rule 6.38. Notice that the copepod tries to avoid being spotted by swimming downwards when it is momentarily in brighter light (rule 6.42); the brighter the light, the faster is downward speed, as shown particularly on the first day, where it began very near to the surface.

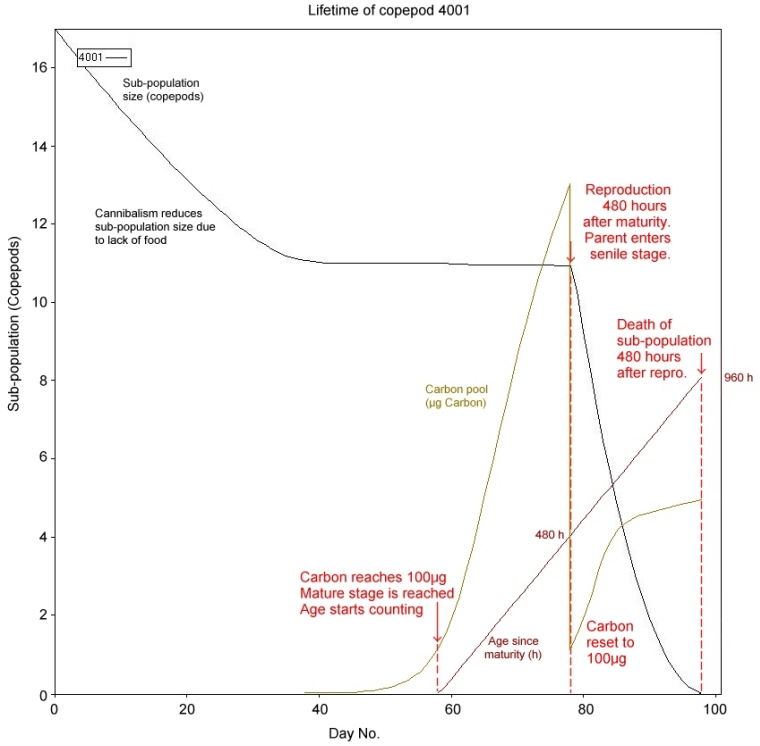

Figure 8.6: Seasonal variation of a copepod

Figure 8.6 shows the life-cycle of a copepod, from juvenile to senile, and death. It begins as a juvenile, and when its carbon pool reaches 100μg, it becomes an adult (rule 6.71). The model assumes that an adult copepod reproduces exactly 480 hours after reaching adulthood (rule 6.73), which is implemented by counting the subsequent timesteps in variable A_r (rule 6.72). After 480 hours, the copepod reproduces, and the carbon the parent gained since maturity is assumed to be distributed among the offspring; the carbon pool of the parent is reduced to 100μg (rule 6.64).

A number of newborn copepods are created at this point, but the parent enters its senile stage (rule 6.78). The 'death by senility' (rule 6.79) causes the sub-population size to reach zero precisely 480 hours after reproduction.

Figure 8.7 shows a 6 year run of live diatoms (the cyst stage is not shown). The system stabilises after an initial transient of around three years.

Figure 8.8 shows the corresponding 6 years of copepod populations - here, the juvenile stage is shown. Although the annual population cycle appears to be stabilising towards the end of the run, it is clear that an equilibrium has not yet been reached.

Figure 8.9 shows a comparison between the original WB model, and the new implementation. As before, the new graphs only show live (non-cyst) diatoms and juvenile copepods. There are differences between the graphs, but this is to be expected, since WB has been reinvented with many changes to fit the Lagrangian Ensemble metamodel. Furthermore, the original WB run shown on this graph is using 100 particles per metre, whereas the new model is only using 20. Notice that the characteristic of a large peak in the first year for diatoms is consistent with the original results, and while the placement of the spring and autumn blooms is slightly different, the shape of the graphs shows similar traits.

The copepod population is consistently lower than results of the original WB model, and seems to be dropping slightly annually after the second year. This may be a stabilisation issue, but certainly needs further investigation.

Further graphs from the original WB model are available for comparison in the literature [56, 59].

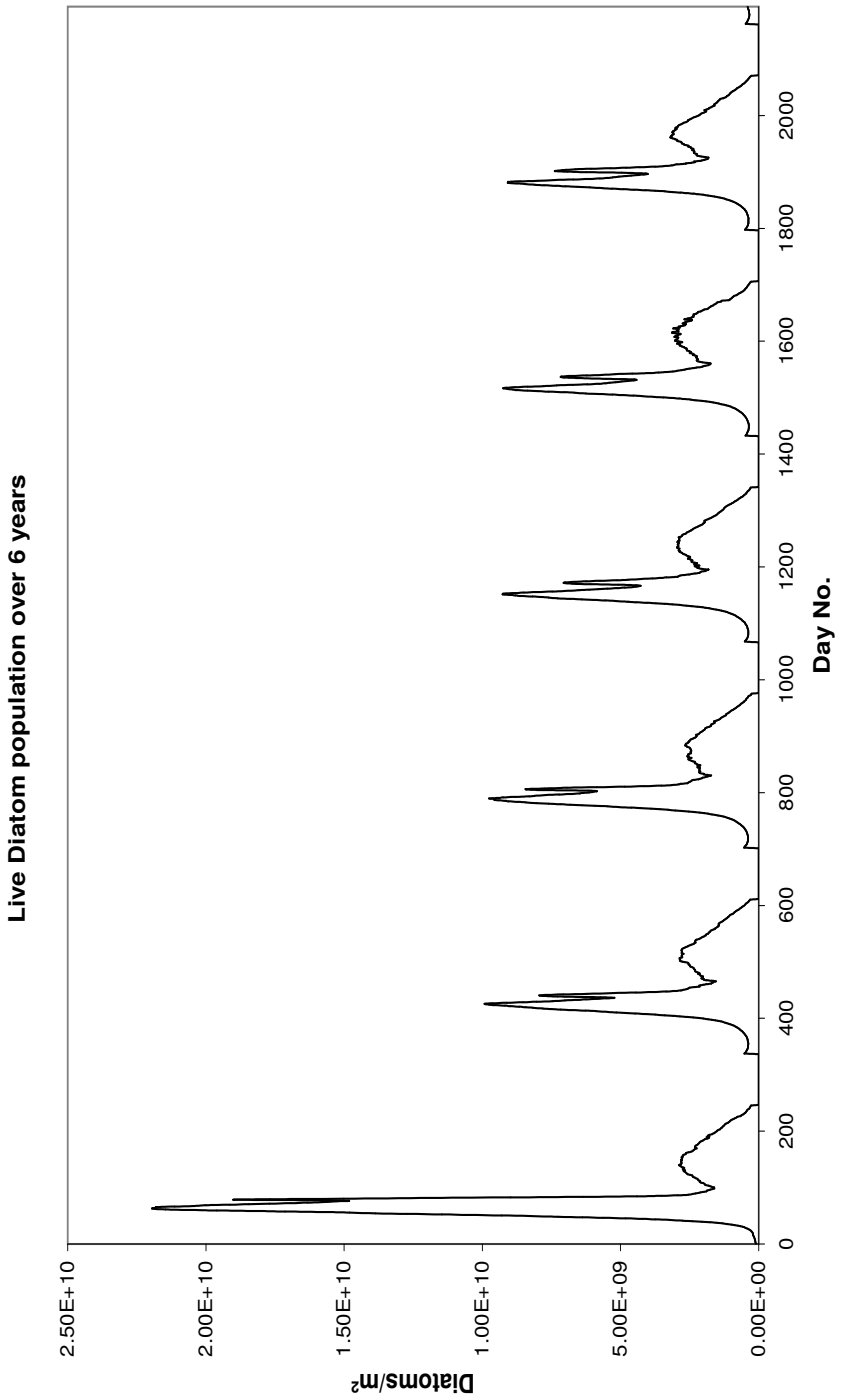

Figure 8.7: Six-year variation of live diatom population

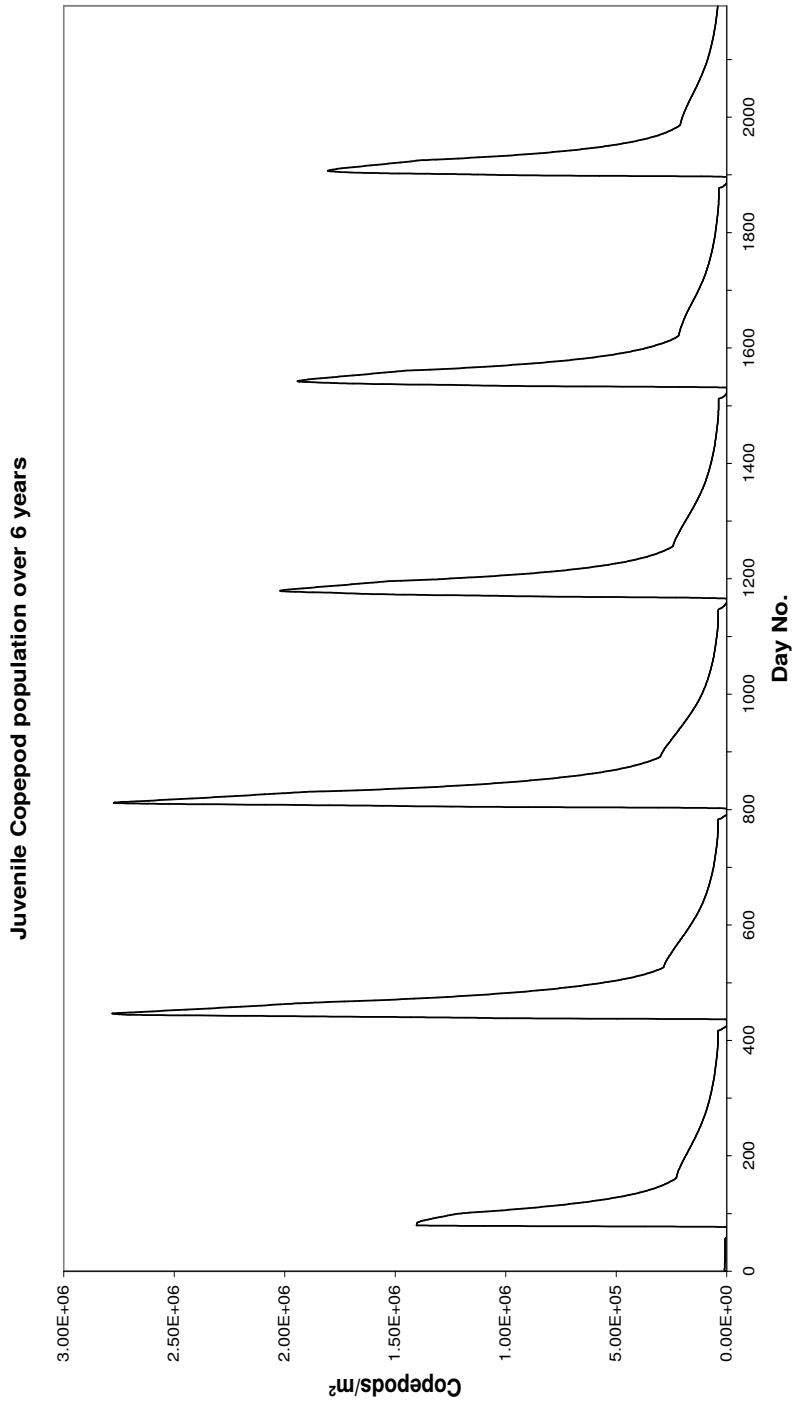

Figure 8.8: Six-year variation of juvenile copepod population

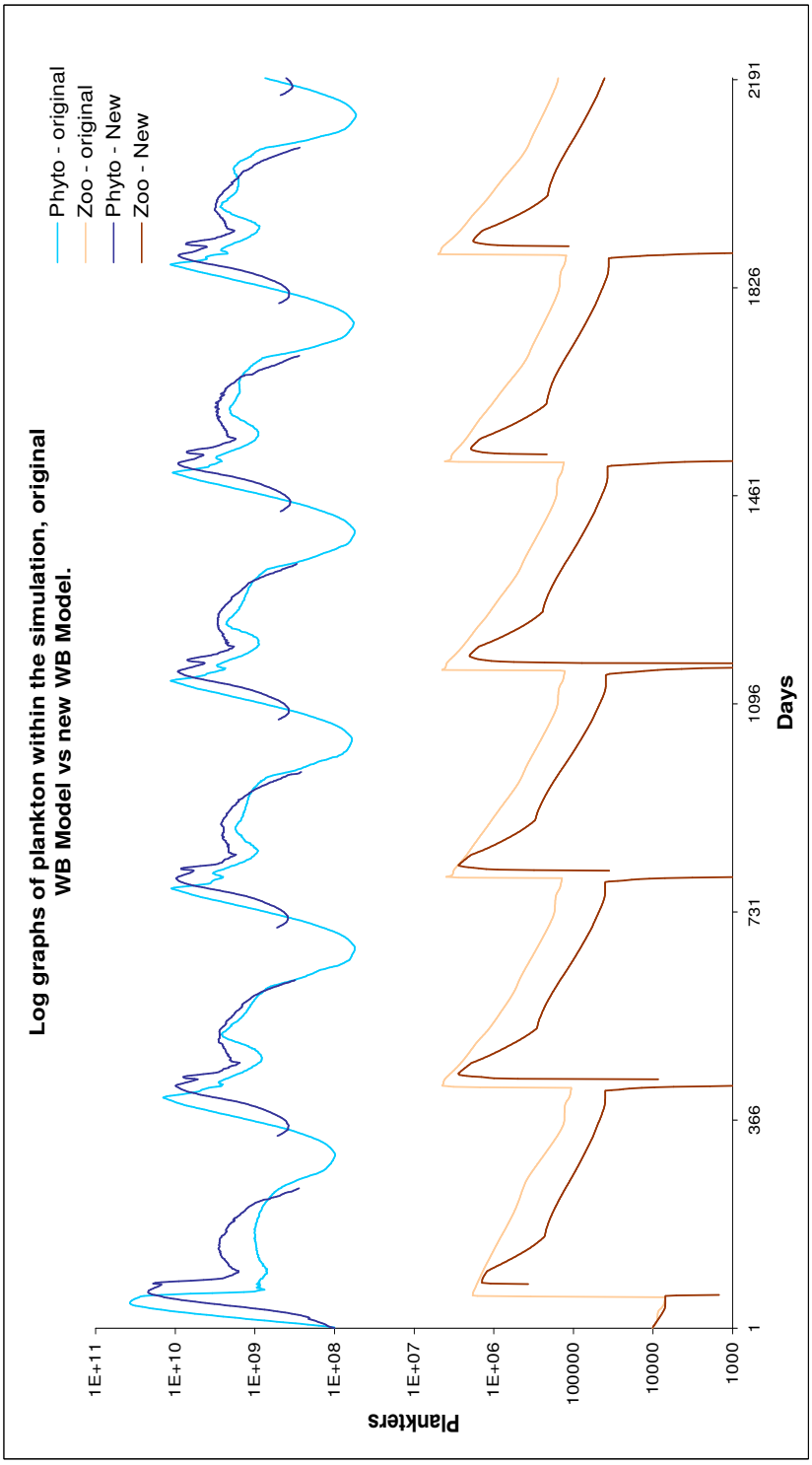

Figure 8.9: Population cycles for old and new WB model,compared

8.2 Modelling Oil Spillage

The significance of modelling the effect that ecological disasters such as oil slicks have on the ocean ecosystem needs little justification [22]. Here, the WB model is extended with a simple representation of an oil slick, which is a single sub-population of oil droplets. Droplets float (rather than being affected by turbulence or sinking as diatoms are - see rules 6.2, 6.4 and 6.3), and occlude light - an effect modelled by setting an individual's pool for a particular pigment.

Over time, the oil droplet is dispersed at a fixed rate; the method of dispersal, whether by human intervention, bacterial breakdown or any other process, is not modelled explicitly. Note that this is not a formal scientific study of the biological processes, but an evaluation of Planktonica's model-building capabilities

The purpose is to demonstrate the ease with which a functional group, a pigment, and rules for oil dispersion can be introduced using Planktonica, and to demonstrate the effects of occlusion caused by the oil. We would expect diatom photosynthesis to be reduced but longer term impact on the diatom population cycle is less obvious.

8.2.1 Implementation

A new functional group called *OilDroplet* is created. An entire slick is modelled as a single Lagrangian Ensemble sub-population of droplets. Oil droplets can exist in one of two stages, *active*, and *inactive*. When a droplet is active, it is defined to contain 1μg Oil, whereas when it is dispersed, it contains no oil; biofeedback depends on the oil content of the droplets, and the pigmentation of the oil (see section 6.3).

When the slick is introduced, all the droplets are in their active stage, but over time, droplets are assumed to be degraded in some unspecified way, and a proportion, d in each timestep move from the active to inactive stage. Particle management then merges the inactive droplets into one sub-population. Hence, during the simulation there will be precisely two Lagrangian Ensemble sub-populations of oil dropets, one of them containing all the active droplets, and the second all the inactive droplets.

Oil pigmentation

A new pigment (a chemical with action spectra), *Oil*, is created. The action spectra are shown below; they differ from chlorophyll in that they affect visible light equally across all wavelengths. Note that the purpose is not to model precisely the pigmentation properties of oil; the spectra below are contrived to clearly demonstrate the biofeedback effect.

w (nm)	300	357.5	387.5	412.5	437.5	462.5	487.5	512.5	537.5	562.5	587.5
χ	0.0	0.0	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
e	1.0	1.0	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8
w (nm)	612.5	637.5	662.5	687.5	712.5	737.5	787.5	900	1100	1300	1500
χ	0.1	0.1	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
e	0.8	0.8	0.8	0.8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
w (nm)	1700	1900	2100	(2300)							
χ	0.0	0.0	0.0								
e	1.0	1.0	1.0								

Figure 8.10: Action spectra for oil biofeedback

Active Oil Content

While the oil droplet is active, it is defined to contain 1μg of oil pigment. As described in section 6.3, biofeedback depends on the concentration of all chemicals within the particles, and the associated action spectra for each chemical.

$$Oil_{pool} = 1 \tag{8.1}$$

This rule could equivalently have been specified as an initialisation property of the oil using the initialisation options in VEW Controller (see section 7.4), since it remains constant throughout the lifetime of an active oil particle.

Dispersion

The dispersion model is very simple: a droplet has a certain probability, d, of being dispersed, whereupon it ceases to act as an occluder. This is applied to the sub-population using the *pchange* function (see section 3.4.1). The proportion d of the sub-population is set to change stage from *active* to *inactive* in each timestep. Three instances of the experiment will be run, setting d to 0.002, 0.0025 and 0.003.

$$pchange(inactive, d) \tag{8.2}$$

Inactive Oil Content

When an oil droplet is inactive, it is assumed that the pigment has been removed by some means, so the amount of oil pigment in the droplet is set to zero, and it has no further biofeedback effect. The assignment:-

$$Oil_{pool} = 0 \tag{8.3}$$

is defined to occur in each timestep for particles in the inactive stage.

Scenario

The scenario extends that of the new WB model, by adding a single event (see section 7.5). The event adds one sub-population of 10000 oil droplets in the *active* stage, 35 days into the second year of the simulation, at the surface of the column. Figure 8.11 shows the amount of oil at the surface over time for each dispersion rate, d.

8.2.2 Results

The immediate effect of the oil slick is the turbidity of the water increases near the surface as the oil cells absorb the sunlight. Figure 8.12 shows the visible irradiance at noon each day. Compare this to the unaffected visible irradiance in figure 8.1. Notice particularly that the turbidity of the water in figure 8.1 is greatest shortly after the beginning of the year, whereas on figure 8.12, this fluctuation happens after the interruption in the

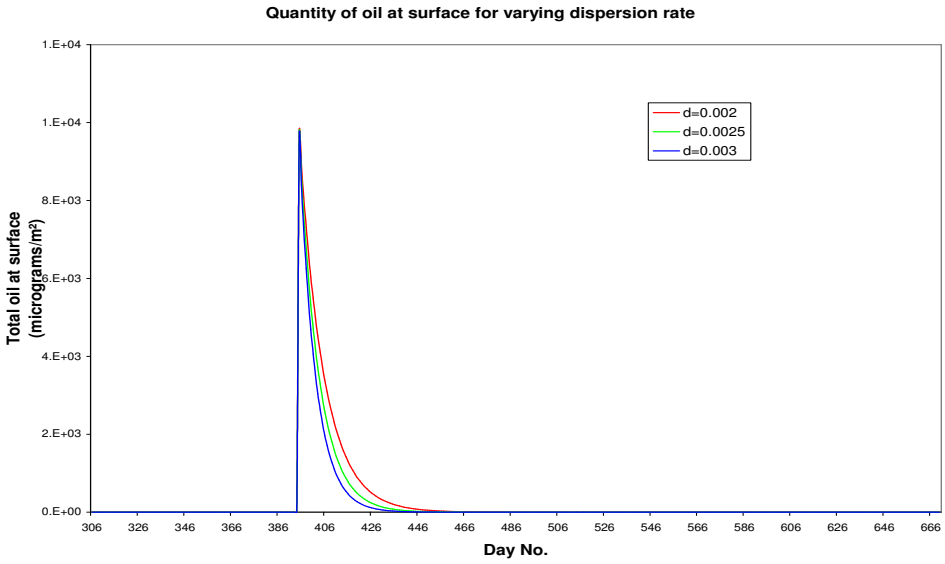

Figure 8.11: Oil Slick content in column

irradiance due to the oil slick. In both cases, this coincides with the timing of the diatom bloom.

The effect of the oil is that the spring bloom of diatoms is pushed substantially later, shown in figure 8.13. This is because the diatoms are not getting enough light to increase their energy by photosynthesis (rule 6.9), and their energy pool does not reach the threshold for cell division (rule 6.15). When the bloom finally does occur, the diatom population rises steeply; the diatoms have gained less energy by photosynthesis, but their nutrient pools will not have been affected. Note that the graphs of diatoms only show their live stage; cyst diatoms in the winter months are not shown, explaining the absence of any diatoms at regular intervals.

The effect of the slick on the future years of the diatom population is shown in figure 8.14. Curiously, the spring bloom of diatoms in future years is generally larger than

the control graph, whereas the autumn bloom is consistently lower for all experiments affected by the oil slick. A possible explanation for this emergent behaviour is as follows:-

- Oil spillage occurs. Diatoms fill their nutrient pools, but the oil restricts photosynthesis, and their energy does not reach cell division threshold.
- As a result, copepods have less food, and their population is lower.
- When the oil disperses enough for the energy pools to rise, the diatoms divide rapidly since their nutrient pools are full, and the copepod population is smaller.
- As the diatom population rises rapidly the copepod population also rises rapidly, as they now have a higher concentration of food than in 'normal' years.
- Since the copepods reached adulthood later due to lack of food, they also reach senility and death later. Therefore, there is a higher concentration of copepods in Autumn than in normal years, which makes the Autumn bloom lower than in 'normal' years.
- As the following spring bloom approaches, there are fewer diatoms, due to the copepods heavier grazing in Autumn. The population therefore rises more slowly, and there is less food for the copepods than usual in Spring. Hence fewer diatom are ingested by copepods as they bloom, and the diatom bloom is larger than in usual years. The copepod bloom is later as a result, and hence the cycle has perpetuated.

8.2.3 Discussion

These experiments required simple additions to the WB model, taking only a few minutes to implement. The emergent behaviour exhibited by the experiments, however, would take substantially longer to analyse and explain. A wide range of experiments can be conducted extremely easily by altering the parameters and pigmentation, or the time of year at which the oil slick occurs, and its initial properties.

More interesting experiments could involve the breakdown of the oil, which has been approximated to a constant dispersion parameter. The dispersion process in the model simply causes the occluding pigment to be 'turned off'. It would be straightforward to model the chemical breakdown of the oil and its effect on the ambient chemistry by adding new chemicals and new rules for the associated chemical processes. This would be similar to the rules for remineralisation of detritus and faecal pellets in WB, for example.

In practice, oil is broken down by a range of physical, chemical and biological processes [31]. In principle, these could all be described in some way with suitable extensions to the model.

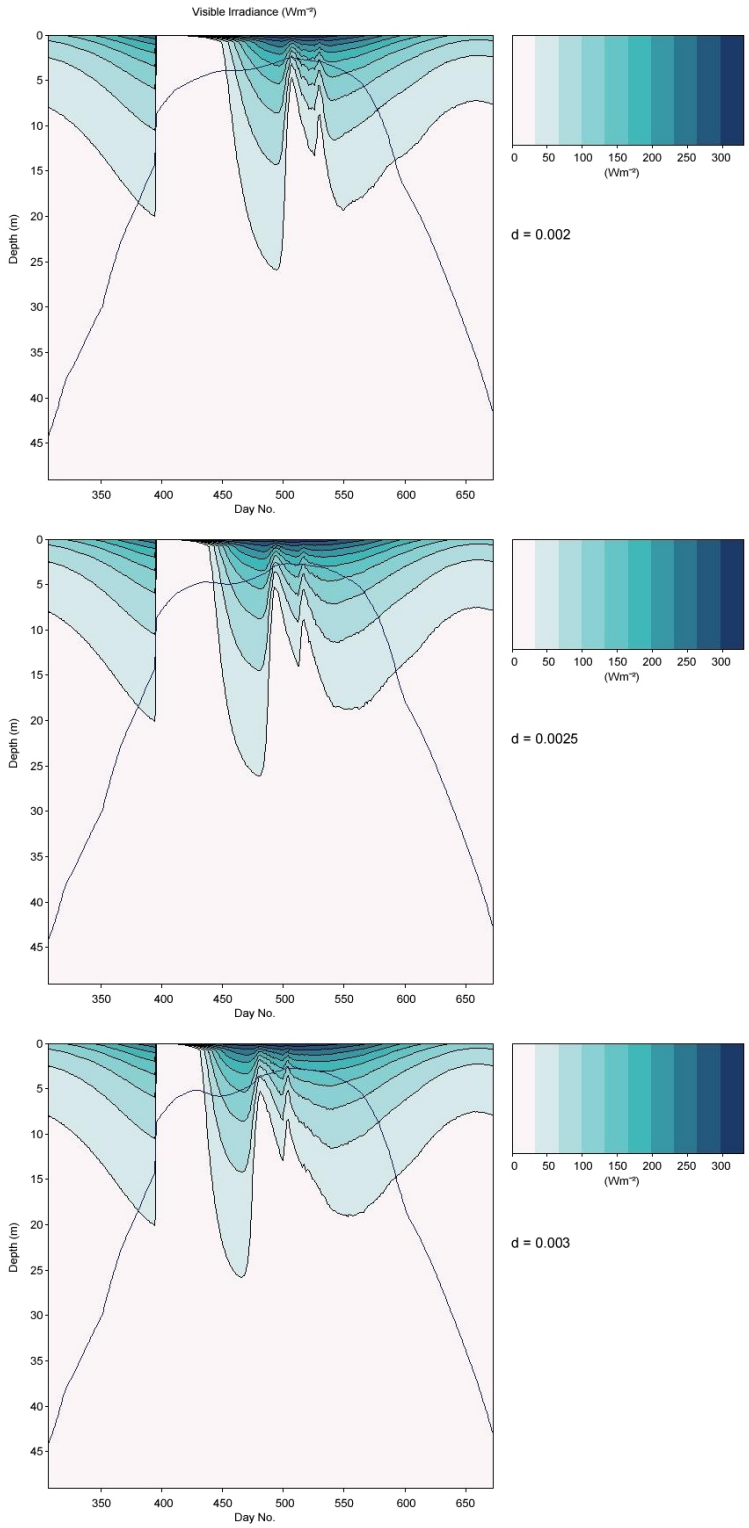

Figure 8.12: Effect of oil on visible irradiance

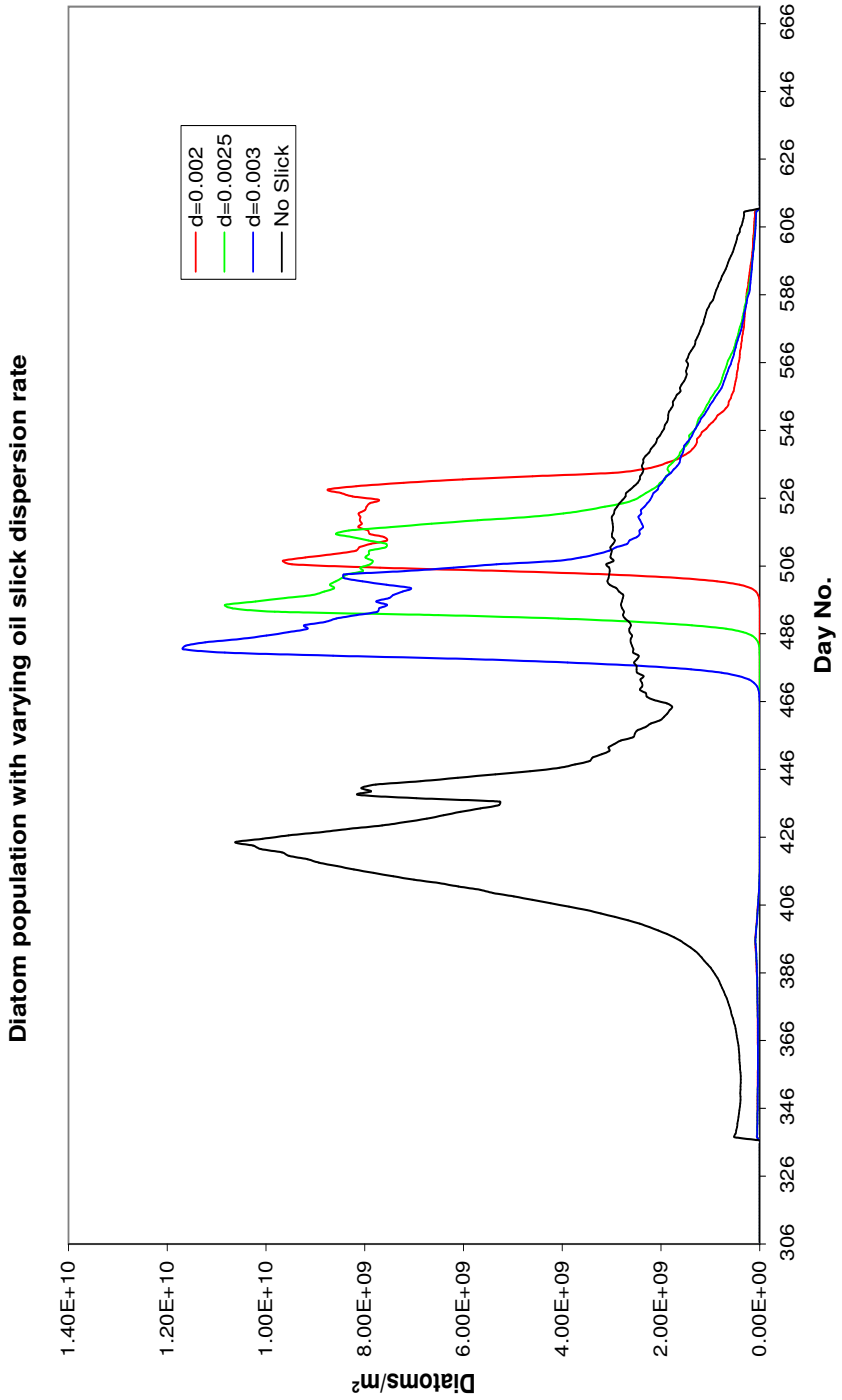

Figure 8.13: Diatoms in second year for different dispersion rates

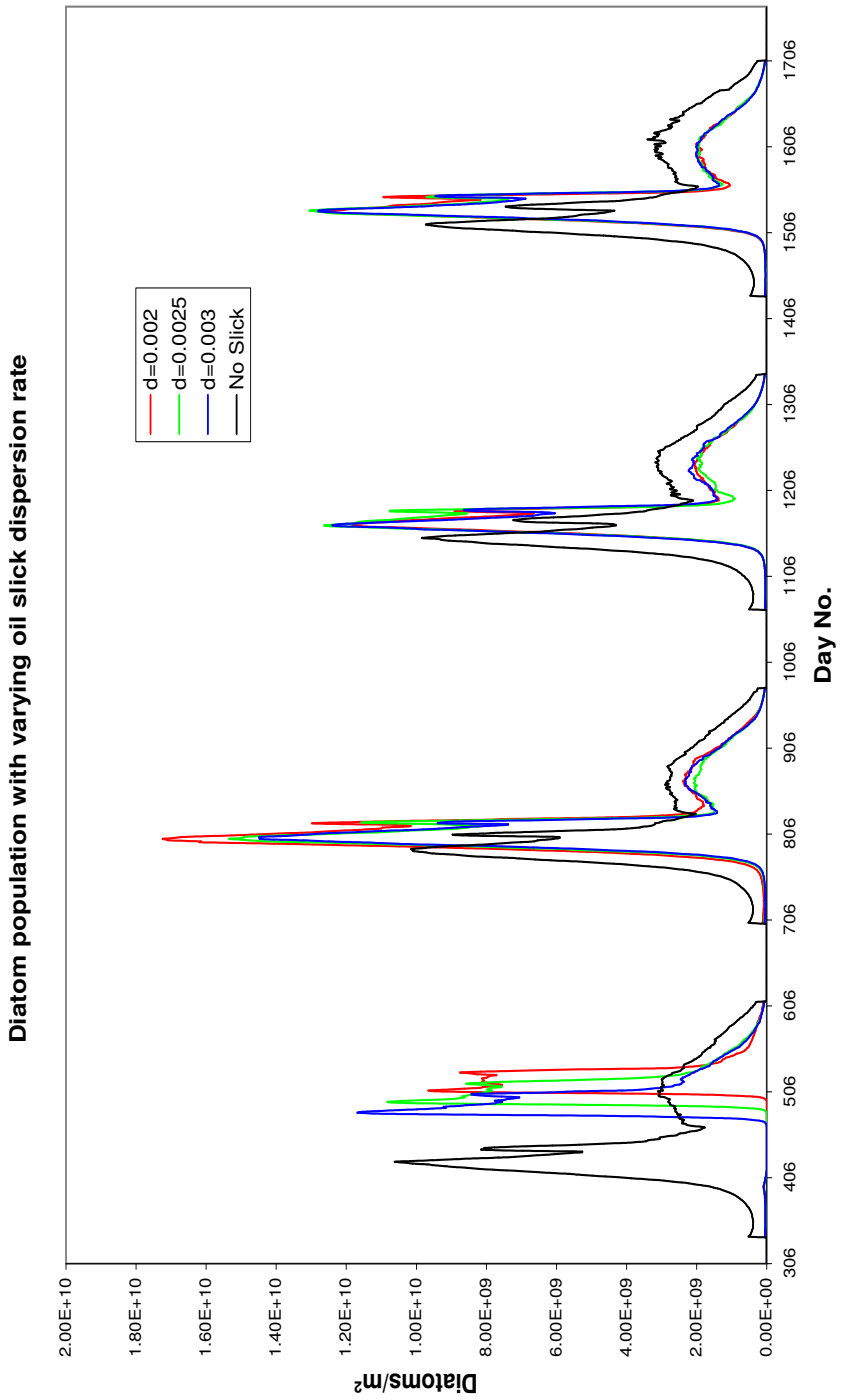

Figure 8.14: Diatoms from second to fifth year for different dispersion rates

8.3 Gut-passage time

This experiment implements work by Simon Smith [47], in which a time delay was introduced for copepods, between ingestion and excretion. This was a project conducted over one summer, modifying the C version of the WB model.

In the unmodified WB model, copepods excrete faecal pellets, the contents of which depend on what the copepod has eaten in that timestep. The gut-passage experiment introduces a time delay between ingestion and excretion, so that the copepod in one timestep excretes pellets that contain what it ate some time ago. If the copepod has migrated during that time, then the associated pellet will be produced at a different location. In this way, copepods are able to transport material from one location to another. This includes nutrients that are released back into solution by remineralisation, but may also include additional chemicals in the environment, e.g. pollutants or toxins, previously ingested by the copepod.

8.3.1 Implementation

A chemical P, representing an abitrary pollutant is introduced, with initial concentration zero throughout the column. The particles inherit P_{pool}, P_{uptake} and P_{ingest} as usual. At midnight on day 90 (1^{st} May), 50 μgP is added at depth of 25m. This is below the deepest location of the turbocline, which stays above 20m between May and November.

Diatoms that have dropped below the turbocline will sink, uptake the pollutant at a rate u_P, up to a maximum content of p_{max}. Copepods will ingest them, and then swim upwards for food, excreting the pollutant g_t hours later.

During the time g_t, copepods may swim upwards to feed on diatoms, or they may also ingest the diatoms on the way down to deeper water. The gut-passage time may even be long enough for them to swim upwards to feed, and then return to deeper water before excreting the pollutant. Therefore pollutant will be distributed above and below the turbocline.

Diatom Uptake

The maximum amount of pollutant a diatom can contain is set at $p_{max} = 1.16\text{x}10^{-7}$ μgP. Its maximum rate of uptake is set at $u_P = 4\text{x}10^{-10}$ μgPh^{-1}. A local variable p_{pot} ($\mu g\, P$) is created and used to store the maximum amount of pollutant that can be absorbed, before p_{max} is exceeded.

$$p_{pot} = p_{max} - P_{pool} \tag{8.4}$$

The diatom then uptakes at its maximum rate u_P, converting from per-hour into per-timestep, unless it will exceed its maximum content p_{max}, in which case the rate is reduced accordingly.

$$if\ (p_{pot} > 0)\ then\ uptake(min(u_P \triangle t, p_{pot}), P) \tag{8.5}$$

As before, this is subject to the availability of the pollutant, and depletion handling. The amount obtained in the previous timestep is stored in P_{uptake} as before. The pool must be adjusted accordingly:-

$$P_{pool} = P_{pool} + (P_{uptake} \triangle t) \tag{8.6}$$

Diatom Remineralisation

Dead diatoms are assumed to remineralise pollutant in the same way they remineralise ammonium. Hence, two rules are added to the diatom remineralisation function, in which P_{remin} is the remineralisation rate, here 0.00208333 μgPh^{-1}.

$$release(P_{remin} P_{pool} \triangle t, P) \tag{8.7}$$

$$P_{pool} = P_{pool} - (P_{remin} P_{pool} \triangle t) \tag{8.8}$$

Copepod Growth

When copepods ingest diatoms, the P_{pool} of the diatoms will automatically be placed in the P_{ingest} state variable within the copepod. A single rule is added in the growth function to assimilate the pollutant into the pool:-

$$P_{pool} = P_{pool} + P_{ingest} \tag{8.9}$$

Copepod Excretion

A state variable, P_h is created; it has an associated history that will store the pollutant ingested over a specified number of previous timesteps. The experiments conducted will test a history size of up to 4 hours. The history size, measured in timesteps, is set to 9; this implies P_h can hold the value of the current timestep, and an additional 8 previous steps. P_h is set as follows:-

$$P_h = P_{ingest} \tag{8.10}$$

This assigns the value of P_{ingest} from the previous timestep to the value that P_h will have in the next timestep.

The pollutant is now remineralised over time. A paramter g_t (hours) is defined representing the time between ingestion and excretion. For the experiments here, we set this to 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5 and 4.0 hours in turn.

$$release(P_{h[\frac{-g_t}{\triangle t}]}, P) \tag{8.11}$$

The subscript of P_h defines the index of the timestep to be retrieved, relative to the present. For example, if g_t is 0.5 hours, then dividing by the timestep gives -1, meaning a single timestep ago. Finally, the pool must be reduced:-

$$P_{pool} = P_{pool} - P_{h[\frac{-g_t}{\triangle t}]} \tag{8.12}$$

8.4 Results

Figure 8.15 shows the initial injection of pollutant at 25 metres. It disperses over time as diatoms sink to 25m and uptake the pollutant. Note that the distribution is not visible on this graph due to the scaling; the concentration of the pollutant is $50\mu\text{gPm}^{-3}$, whereas the concentrations when distributed are much smaller, as shown in later graphs.

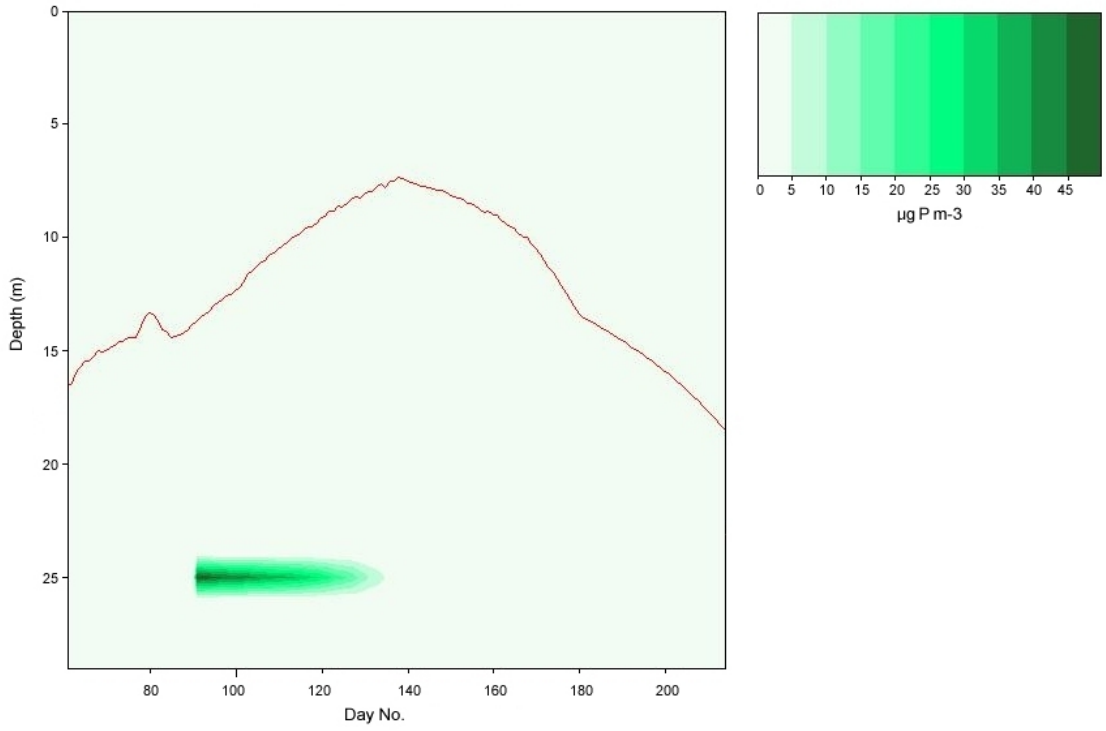

Figure 8.15: Initial injection of pollutant, (not distribution)

Figures 8.16, 8.17 and 8.18 show the distribution of the pollutant for each gut passage time. For each gut passage size, the left-hand graph shows the concentration above the injection depth and turbocline, whereas the right-hand graph shows concentration below the injection depth. Note that the scales of the graph vary, as the concentration of pollutant occurs in patches of variable concentration.

In general, as the gut passage time increases, the pollutant concentration between

about 60 and 100 metres increases. Referring back to the graph of copepod migration, figure 8.5, this is logical, since copepods spend more of their time in deeper, darker water to avoid predators, than near the surface. If their gut passage time is shorter, then there is a higher likelihood of excreting the pollutant nearer to its source at 25m.

The pollutant concentration above the turbocline when gut passage time is 1 hour, is substantially less than if g_t is increased to 1.5 hours. This is also explained by diel migration; copepods only venture near the surface, risking predation, if they are sufficiently hungry. On their way to the turbocline, they pass through the source of pollutant, and ingest sinking diatoms that have been uptaking pollutant. If their gut passage time is long enough, then they will be able to swim up to ingest on diatoms near the surface, and down again, before the pollutant they ingested on the way up, is excreted.

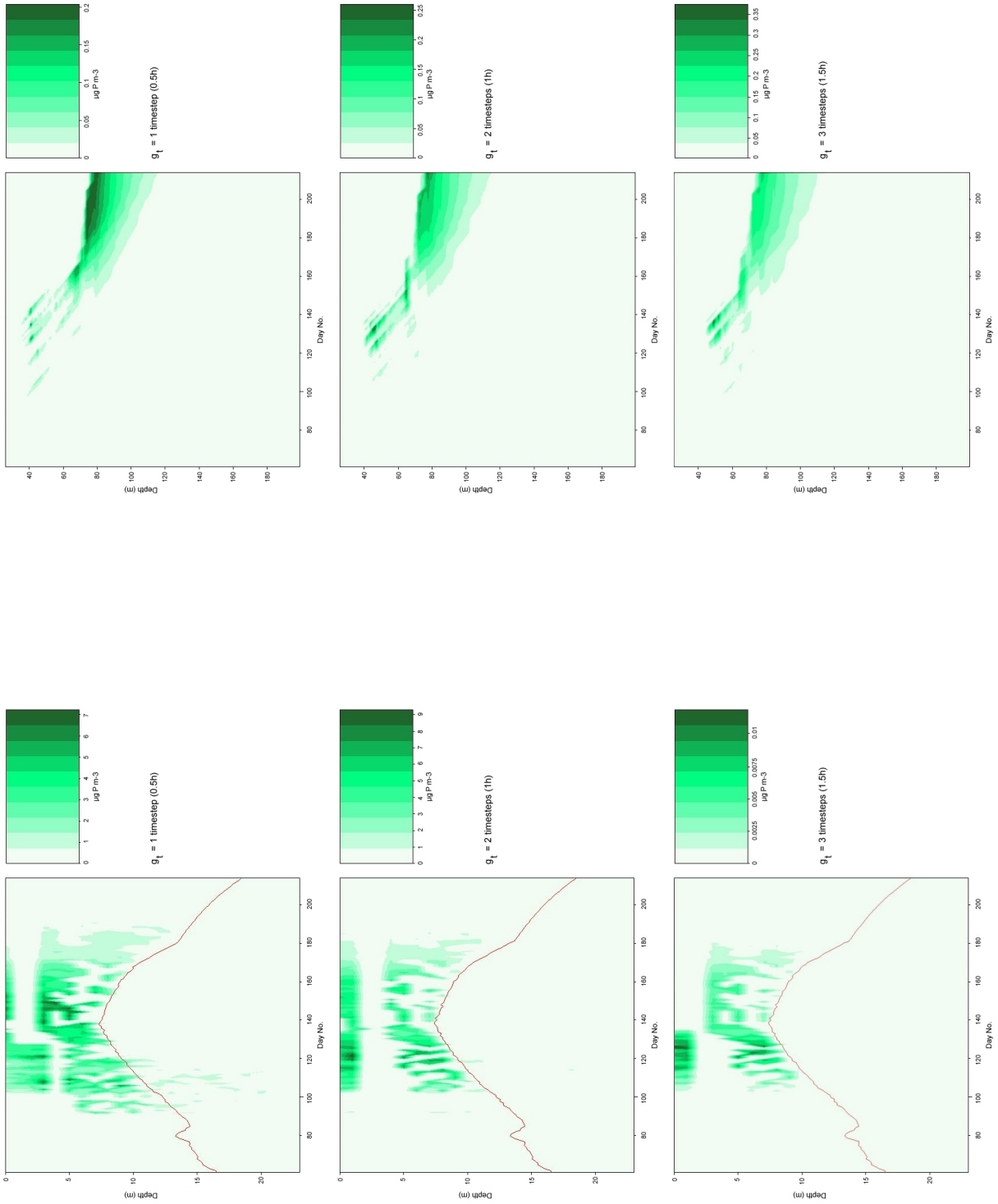

Figure 8.16: Pollutant distribution (1), above and below the injection point

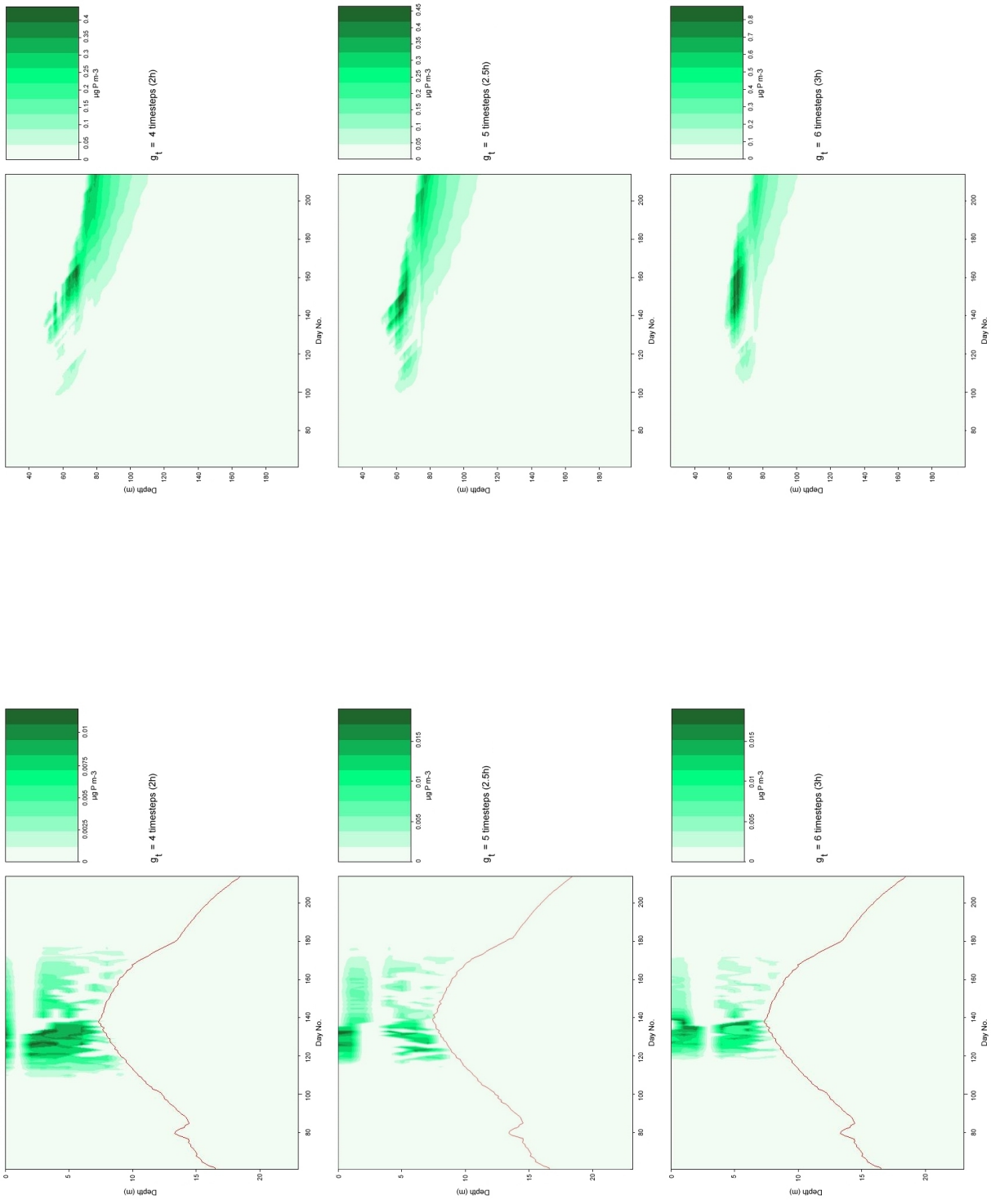

Figure 8.17: Pollutant distribution (2), above and below the injection point

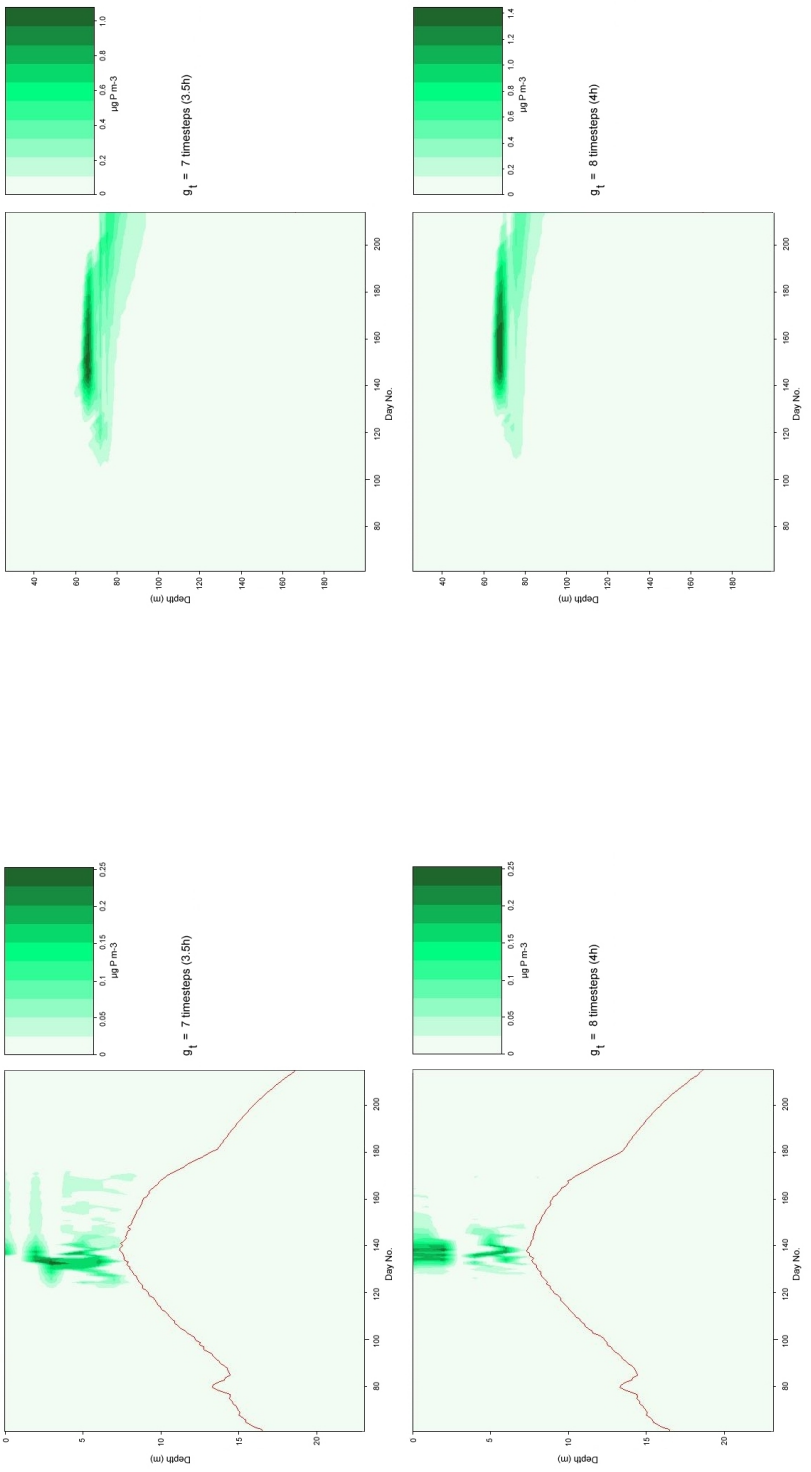

Figure 8.18: Pollutant distribution (3), above and below the injection point

8.5 Discussion

In this experiment, pollutant was inert, and did not affect the particles, but the principle of copepods transporting chemical is interesting. In the case of ammonium, which becomes depleted at times of year shown in figure 8.2, copepods with a gut-passage time may transport ammonium from deeper water to the surface, supplying more nutrients for diatom cell division [8].

The rate at which disease spreads in water is also related to the gut passage time. Such diseases could affect the plankton [5], or they could be diseases that affect humans, such as cholera [45].

All of these experiments are easily built in Planktonica; the gut passage experiment demonstrates the behaviour of variables with associated histories, which make the construction of the experiment here possible in about 15 minutes, compared to several weeks of project work implementing changes by hand to the C code [47, 8].

Chapter 9

Conclusions

In this chapter, the success and limitations of Planktonica as a modelling environment are summarised, and future improvements and applications are described.

9.1 Summary and Reflections

9.1.1 Model building pre-Planktonica

The code in question was the original WB model, which was built in modules, but the interactions between them were complex. For example, the presence of chlorophyll in diatoms was embedded into the physics in a way that made separation of biology and physics complicated. Adding a new type of particle with a new pigment would require a software engineer to unpick the way that chlorophyll was integrated, and then implement one of two options. The first, and the most forward thinking would be to implement a dynamic data structure so that any number of pigments could be added, and reimplement chlorophyll, and the new pigment using that structure. The alternative would be to duplicate the chlorophyll code to get the required behaviour in place as quickly as possible.

Unfortunately, in many cases, the technical challenge of creating a highly flexible and extendable system, combined with the time taken understanding the tangled structure that already existed, proved too much for short-term project students, who contributed

largely to the development of the old WB model, albeit in small enhancements and additions that would never be properly incorporated, or built upon.

9.1.2 The design of Planktonica

The design of Planktonica was established by taking the C code for a one-off simulation, separating the *generic* elements of the code from the *model-specific* elements of the code, and then rewriting the generic parts so that they support any configuration of model-specific elements that could be reasonably imagined.

For example, where the original WB model contained complex code to describe how copepods ingest diatoms, Planktonica was designed to support the *generic behaviour* that any type of plankton may ingest any other type of plankton. This facilitates the construction of complex food webs.

This generates a separation between metamodel and object model, and allows the user to think of the biology of a model completely separately from the physics and chemistry. This simplifies not only the implementation of models, but also the knowledge and effort required by a single user in designing them.

The use of the Lagrangian Ensemble metamodel requires rules to be written for individual plankters, and forbids particle-to-particle interaction. In the original WB model, both of these rules were broken since some rules were written for individuals, others for sub-populations; when implementing ingestion, complex code for particle-to-particle interaction was required as the copepods ingested the diatoms.

This complexity is no longer needed; all the complexity of handling data structures (including the particle-to-particle interaction for ingestion), all the rules that implicitly require manipulation of sub-population sizes and indeed, everything except the task of writing rules to define the behaviour of an individual plankton, are hidden 'behind the curtain'. Where access to variables kept behind the curtain is required, 7 simple API calls are provided.

9.1.3 Building Models

Models in Planktonica are built 'in front of the curtain' without the user ever having to worry about the housekeeping code beneath. This makes the process very much easier, since the user only has to think about the biological and chemical processes of the individuals being described.

This is in contrast to previous versions of the VEW, where the user would, for example, need to understand the inner workings of the physics code, in order to implement biofeedback, and face the likely risk of introducing a separate bug in the physics.

Furthermore, a model is now built out of components that are based on nature: functional groups, species, varieties and chemicals, with their associated rules. These are abstractions familiar to a biologist, compared with the various underlying data structures that are only readily accessible to a software engineer.

Planktonica can be seen as a system that can build complex models from simple components. A model containing many functional groups and chemicals is no more complex to design, than a model containing just one functional group.

9.1.4 Building Rules

When rules are built, Planktonica attempts to remove as many potential sources of error as possible. For example, it is impossible to reference a variable or a chemical that doesn't exist, since variables and chemicals are selected from menus that offer choices that are valid.

It is not possible in Planktonica to form a syntactically incorrect statement, since the user is guided at each step of building a rule, and can only choose options that make syntactic sense. Similar type-checking options prevent the user from assigning values to constant parameters, for example. Some safety features may seem a little inconvenient, for example subtraction and division are only permitted as binary functions to avoid potential associativity errors. While some may think this unnecessary, it is a very small price to pay to reduce the potential for common mathematical errors.

A common problem in biological modelling is the conflicting use of units, a prime

example being chemical quantities being expression in either *millimols* or *micrograms*. In Planktonica, all 'built-in' concentrations are expressed in μgm^{-3}. This may not be to everyone's preference, and perhaps future enhancements could allow Planktonica to switch to $m\text{Molm}^{-3}$ if the user prefers. The important point is that all units must be consistent, regardless of which unit is chosen. To this end, all variables and values in Planktonica have units defined by the user, and a unit checker is available to verify the consistency of the units in each rule.

Variable Types

Planktonica does have a number of types of variables, which the user must familiarise themselves with. They are not very complicated in terms of software engineering, but they have separate purposes, and cannot be treated like simple mathematical identifiers. For example, local variables are not buffered, but state variables are - see section 4.3; they behave differently when read, or written to. Using them interchangeably will result in the model behaving differently to how the user intends. However, on examination, the different types have different purposes, that are necessary to the system, and correct understanding and use of, for example, exported variables, can create neater models - see section 4.3.5.

The variety-based types (section 4.3.9) are more complex data-types, especially since when writing rules, the user will not have set up their contents yet, thus making their use somewhat predictive and abstract. However, they also offer an elegant way of stating that a particle ingests *any number* of other particles at different rates, using just a single rule. Therefore, if the user can fully understand the motivation for variety-based types, he will find them very efficient in parameterising ingestion.

The Metamodel API

The API defines all the ways that rules in front of the curtain must interact with hidden properties behind the curtain. In the early stages of analysing the C version of the WB model, it seemed that many API calls may be required for many different biological

processes. However, after much analysis of what behaviour is considered 'generic', only seven function calls were required, and their behaviour is on the whole very simple. The *ingest* function (section 3.4.5) requires some explanation, but given an understanding of the variety-based types, its meaning becomes intuitive.

There is always the possibility that some new behaviour in the future will be required that the current API cannot describe. However, in constructing the new WB model, the experiments described in chapter 8, and a model for testing fisheries recruitment, which is also being conducted in Planktonica [46], it has not been necessary to extend this API.

Buffering and Timesteps

Another area which must be understood is the buffering issue, whereby when a rule makes reference to a state variable, the result is taken from the previous timestep. For example, if a user writes rules that say $a = bx$ and $b = ay$, then they may be surprised that the value of a in the second rule is *not* bx, but rather it is the value of a from the previous timestep.

A similar issue with differential equations was mentioned in section 4.4, whereby differential equations have an implicit $\triangle t$, which can cause particular confusion if the equation is already meant to imply a change in a variable during one timestep.

9.1.5 Model Debugging

Nothing prevents the user from writing a rule that contains a divide by zero, or the log of a negative number, for example. As a consequence, the user can assign such an 'illegal' value to a chemical pool, which may be involved in biofeedback. This can cause system crashes, since the physics code that handles biofeedback is sensitive to incorrect concentrations of pigment. As biofeedback affects all the physical properties of the water column, and the turbocline, such errors can rapidly propogate. Since a single particle with an invalid chemical pool can cause such a crash, this type of error can be hard to track down.

This problem needs addressing, and in general model debugging is an area that requires attention. Previously, VEW Analyser [43] was used to debug models after they had been run. This can plot audit trails of all state variables within particles, and these often show the area of behaviour that is not working correctly. However, this requires the simulation to have been run already, and locating the particle with a fault can be very time consuming.

As part of the work described here, LiveSim was therefore created, which enabled a model to be debugged interactively for the first time. Particle state variables, chemical concentrations, particle concentrations and physical properties of the column can be displayed at each timestep. Particles to view can be sorted by any of their properties, for example the heaviest copepod, (and hence the first to reproduce) can be tracked. It also supported snapshots, whereby a simulation can dump its state at regular 'checkpoints', and debugging can resume at any checkpoint. LiveSim proved extremely useful in debugging models, and was extended further by a *debug* option when compiling models, which also enabled the user to view local and exported variables for each particle. It has recently been further modified to improve its presentation and offer a wider selection of graphs [37].

However, even with LiveSim exposing all the variables, the reasons why results do not appear as the user expected can take some while to diagnose. There is scope for further improvements in this area.

9.1.6 The Legacy Problem

Researchers in the past have written many different versions using the common code of the WB model, each one has no indication of where the model came from, what bugs in the common code have been fixed, or how the model was generally composed.

Planktonica goes along way toward solving these problems. Firstly, the 'common code' as it was in the past, cannot be changed by the user; this is fixed into the simulation. Thus, if bugs behind the curtain are fixed, or new metamodel behaviour is added, such changes are to be done by the future developers of Planktonica, not the users who

build models.

The structure of the specification file used for a model distinguishes clearly between functional groups, functions, subfunctions and chemicals. The intention is that any of these can be archived, and new models can be built by importing arbitrary components from a repositry. The present state of development is that the importing of such components is available, but the automatic archival of complete models has not been implemented. The importer ensures internal consistency (i.e., the existence of all necessary chemicals and variables for an imported function to be valid).

As this archival process is developed further, a system similar to a Concurrent Version System (CVS) may be considered, which would allow sharing of model components, and automatic updates to the latest versions. In this way, Planktonica may become a valuable tool to the wider community of biological oceanographers.

9.2 Future Work

The interfaces in Planktonica are considered as prototypes, and improvements to these would aid users further in constructing models efficiently; some of this work is already underway. For example, LiveSim has already been re-implemented [37], with many new features for interactively debugging and demonstrating models.

The physics code, which has been provided as standard for all simulations in Planktonica so far, is being moved towards the front of the curtain, so that the user has a certain amount of control over what physical behaviour is modelled within the column [44]. This improvement is motivated by a new model for optics [27] offering substantial improvements over the existing one. In the future, an environment similar to Planktonica, but applied to the physics of the model, may allow physical oceanographers to design their own models for the physics.

The current metamodel treats the ocean as a 1-D column; other work is ongoing to develop a 3-D metamodel [44], allowing the modelling of animals that can change their position horizontally. Significantly, models that have been built for the 1-D metamodel

will require little change in order to work in the 3-D model, the changes being those relevant to position and motion.

Such modifications will require an increase in resources. While the old WB model in C used parallel-processing via MPI, the new Java code created by Planktonica does not yet have this capability, although certain design decisions such as variable buffering were made noting that they would be helpful to future parallelisation. The issue of performance and memory usage is one that will need addressing as the size and complexity of models increases.

Meanwhile, Planktonica is already being used to test theories of fisheries recruitment [46], and a model is planned to investigate the visual properties of predators [52]. A model of three trophic layers in a food-web model has been unsuccessfully attempted in the past [36]; this model is also a candidate to be implemented in Planktonica.

9.3 Concluding Remarks

In summary, Planktonica offers a revolutionary improvement in building models when compared to writing them in C, and offers a substantial number of features that aim to make rule building as safe and efficient as possible. It is not perfect and there are areas where an incomplete understanding of its design can lead to confusion and error. However, as demonstrated in chapter 8 a good understanding of how Planktonica works can yield useful scientific results extremely quickly. Over a longer time frame, the separation of metamodel from object model, and the provision for archival offer a new way forward for co-ordinated e-science in the field of biological oceanography.

This thesis aimed to establish whether it is possible to construct complex models of plankton ecosystems using only primitive rules. The conclusion is that it is possible, as long as the metamodel and object model are separated correctly, and the interfaces between them are understood.

Appendix A

VEW Designer Interface

This appendix contains screenshots that accompany the process of building a model, such as the Plankton-complaint WB model shown in chapter 6.

When creating rules, a preview of the rule is shown in the bottom of the rule editor, although it is rendered rather more crudely than is shown in the text of the thesis. The text shows the way the equations and rules are output from the VEW, specifically using a LaTeX-based package called *VEW Documenter* [53]. This takes a Planktonica model specification, and produces typeset documentation for that model, pretty much as it is presented here. Note that an improved rule previewer based on VEW Documenter is actually supported in the current version of Planktonica, although at present the rendering process is rather slow.

Figure A.1 demonstrates the addition of a functional group to the model. An importer window is shown which allows either creation of a new functional group, or import of a functional group that has previously been archived. While the data structures for archival are in place, the functionality for automatic archival of model components has not yet been fully implemented.

Figure A.2 shows the process of adding stages to a model. Buttons are available to add, remove, or rename stages. For the currently highlighted function or sub-function, the checkboxes allow the user to specify which stages that rule should be executed in. If the rules selected contain two assignments to the same state variable, (which is forbidden

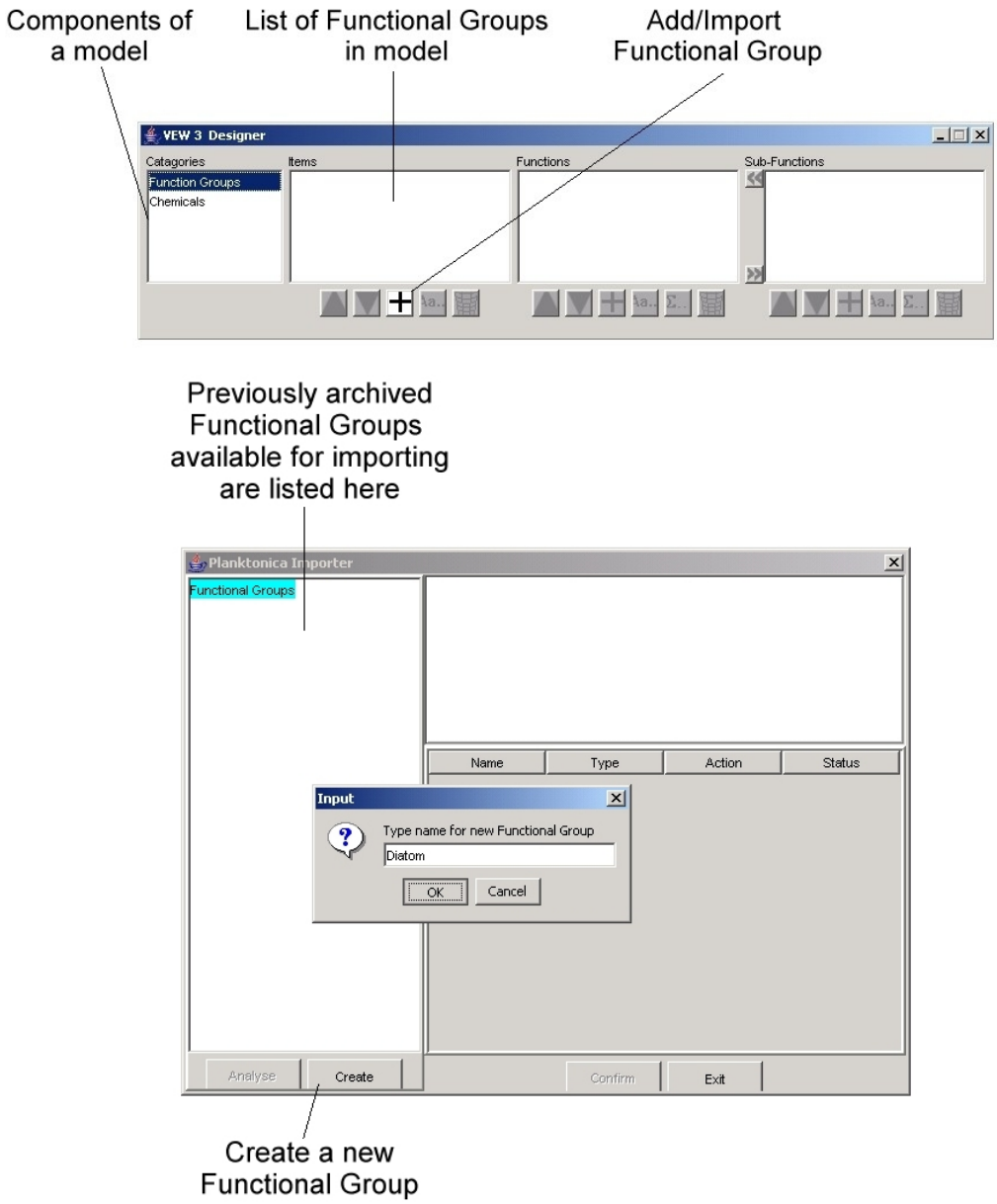

Figure A.1: Adding a functional group

- see section 4.2.1), then an error message is shown and the action is prevented.

Figure A.3 shows the specification of action spectra; the 'chemical has pigmentation' box is ticked, allowing the table on the left to be added. Only the values in the third

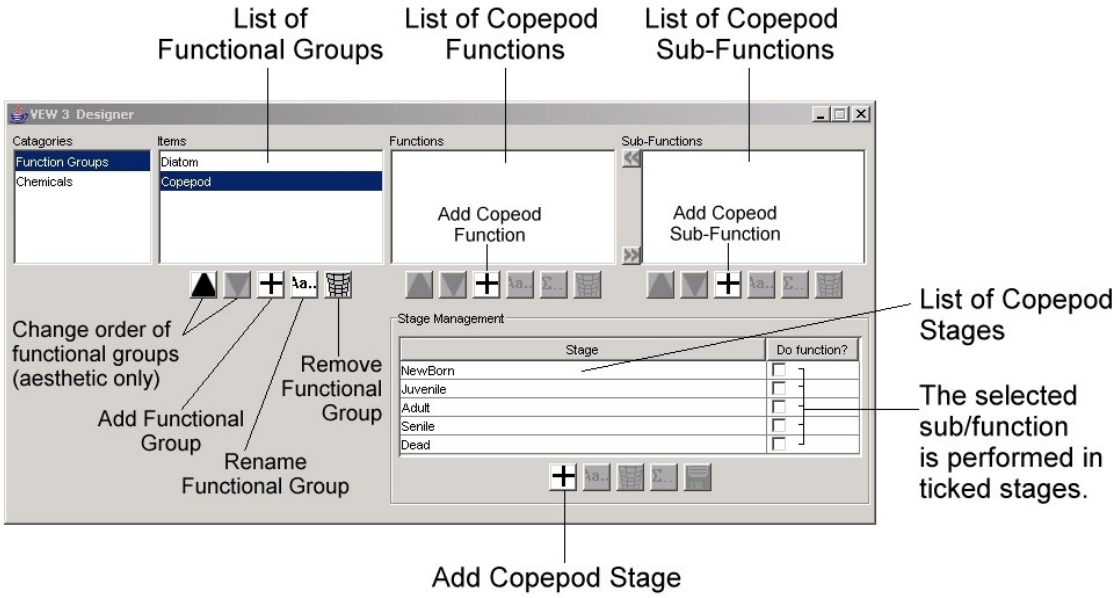

Figure A.2: Adding stages

column can be edited, since the wavelengths are built into the physics code.

A function is created in figure A.4. This causes the Rule Editor window to appear, and rules can be added within the function. To add a sub-function, the corresponding add button below the sub-function list is chosen. Functions can be moved to sub-functions, and vice versa, although if a sub-function contains an exported variable, then attempting to move it will generate an error message, since functions cannot include exported variables.

An assignment is created in figure A.5. The user can navigate through the elements of a rule by either selecting nodes in the tree in the middle left of the screen; clicking on any node or leaf causes the contents of the middle-right pane (the 'detail' window) to update. Clicking on any of the components in the detail window will cause the list in the top right of the screen to update; this list contains the statements, functions or expressions that are available to the user for the element in the detail window they have selected. An add or replace button becomes available as appropriate.

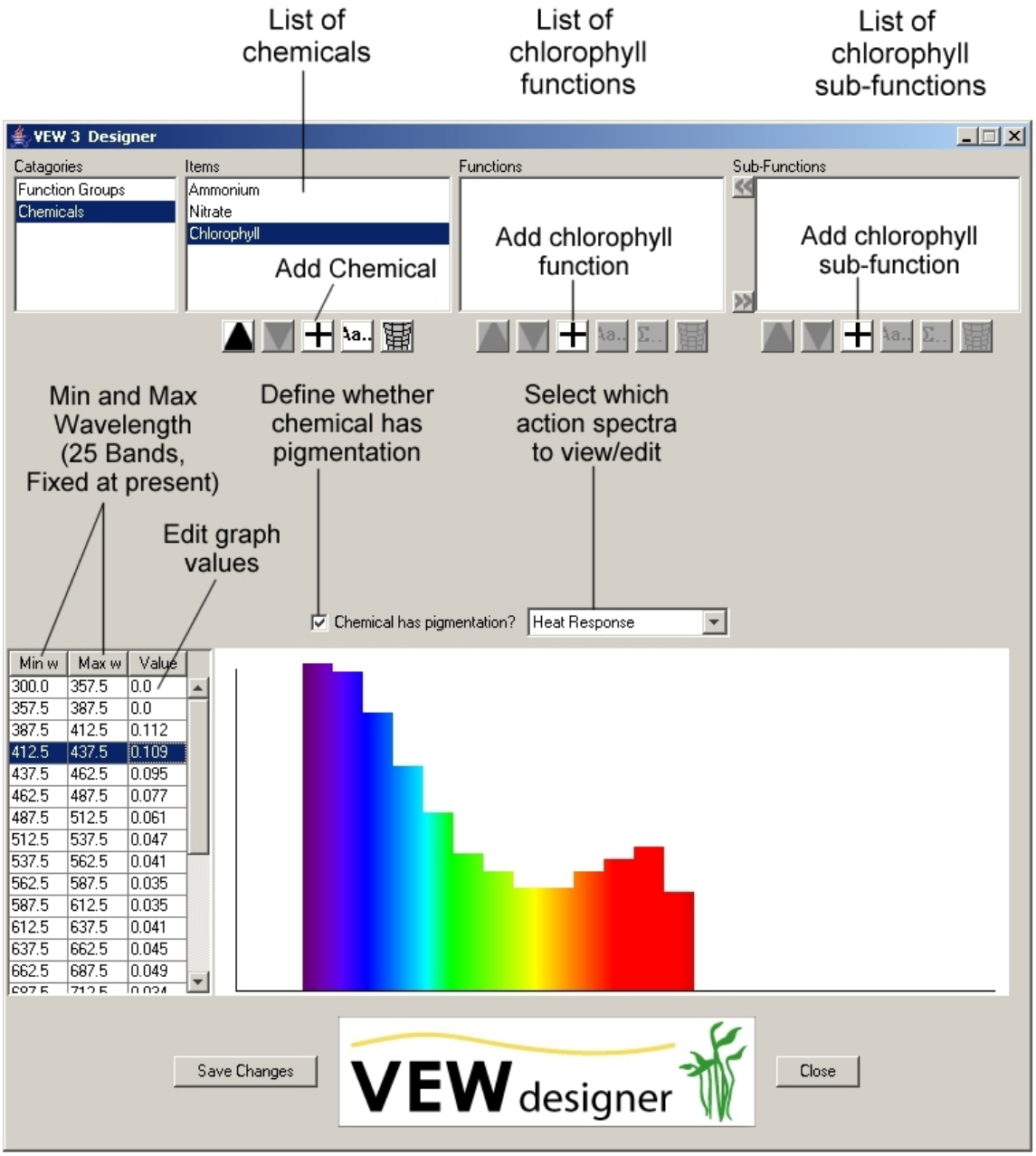

Figure A.3: Defining pigmentation

The surround button allows an expression to be placed around a highlighted element, (e.g. expression e could be surrounded with a cos, to become $cos(e)$. The opposite can

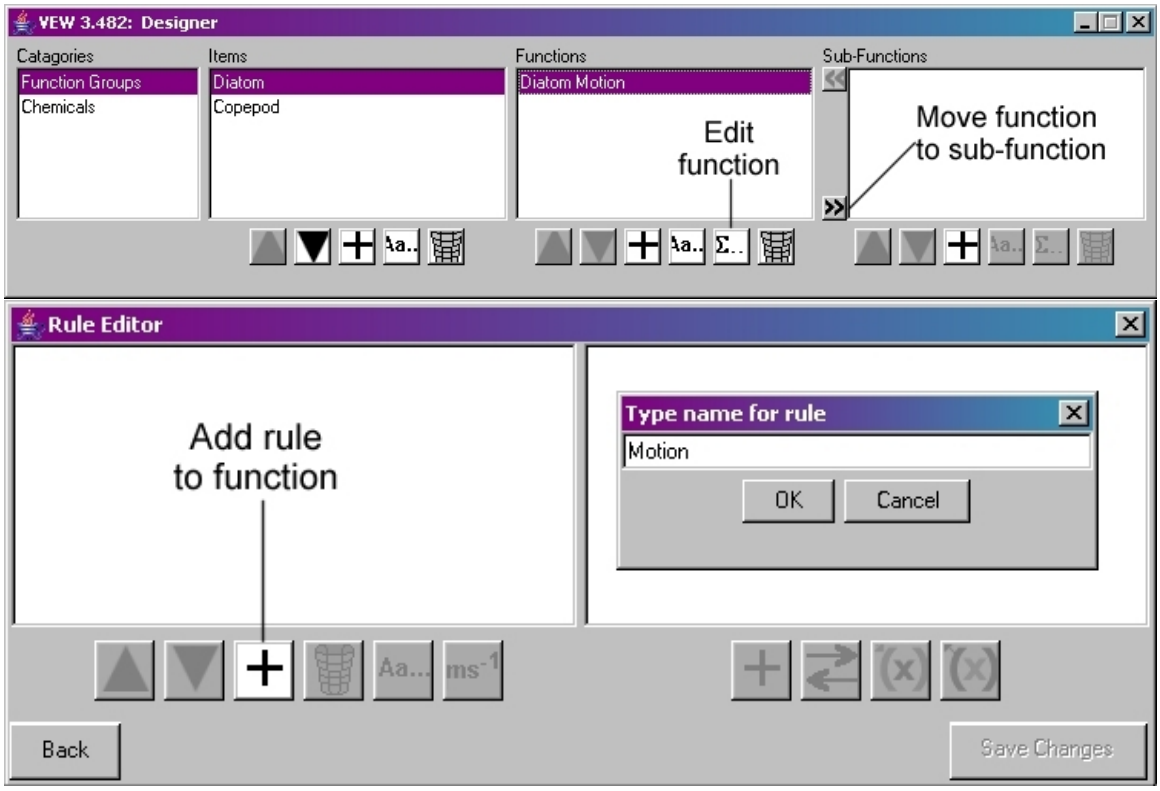

Figure A.4: Creating a function, and a rule within a function

be carried out with the 'remove surround' button, causing $cos(e)$ to return to e.

Selecting an element in the detail window and clicking on the 'expand' button causes the details of the selected element to appear in the detail window, and the selected tree node on the left will be updated appropriately. Many detail windows also allow arguments to be swapped, hence it is easy to swap $a < b$ into $b < a$. In figure A.5 however, the swap button is disabled, since the left-hand side of an assignment must always be a variable, whereas the right hand side can be any numerical item.

Variables are chosen as shown in figure A.6; a menu of all available options for the selected item are shown, separated into categories for convenience.

The 'conditional' expression is selected in figure A.7, and the three arguments when the conditional is edited are shown in figure A.8.

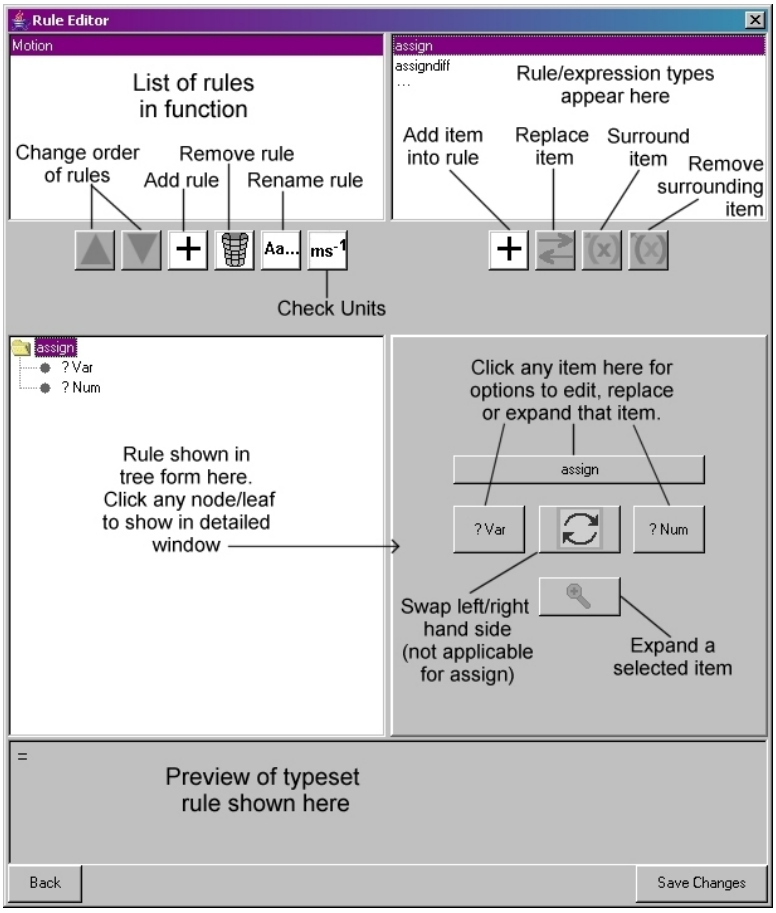

Figure A.5: The Rule Editor, and creating an assignment

An exported variable is created in figure A.9. The variable creation window, top right in the figure, allows the user to set the name, description, default value when appropriate and history size of the variable being created. For parameters, local variables and exported variables, history size is not applicable. For local variables and exported variables, the default value is not applicable.

For variety-based variables, this window is where the user explicitly states that a variable is linked to another one.

Figure A.10 shows a completed rule, in this case motion for diatoms. Note that the window in the bottom shows only a quick preview of the rule being built. It was hoped

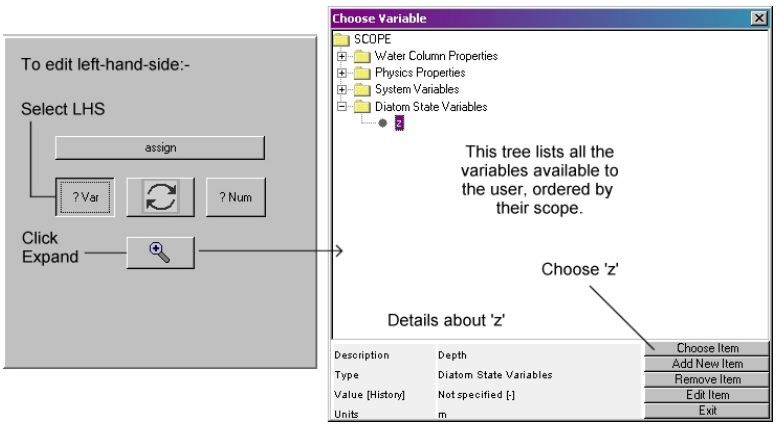

Figure A.6: Choosing 'z' for the left-hand side of the assignment

that a suitable method of showing equations in a typeset way would be available in Java, however no suitable software has been found for this purpose. The preview button launches a window that displays a typeset version of the equation, but it is at present too slow for automatic rendering.

Figure A.11 shows how a numerical value is edited in the VEW designer interface. Note that numerical values must also have units.

Figure A.12 shows how an expression with multiple arguments may be added. Planktonica allows associative operators ('add', 'multiply', 'max', 'min', 'and' and 'or') to take one or more arguments.

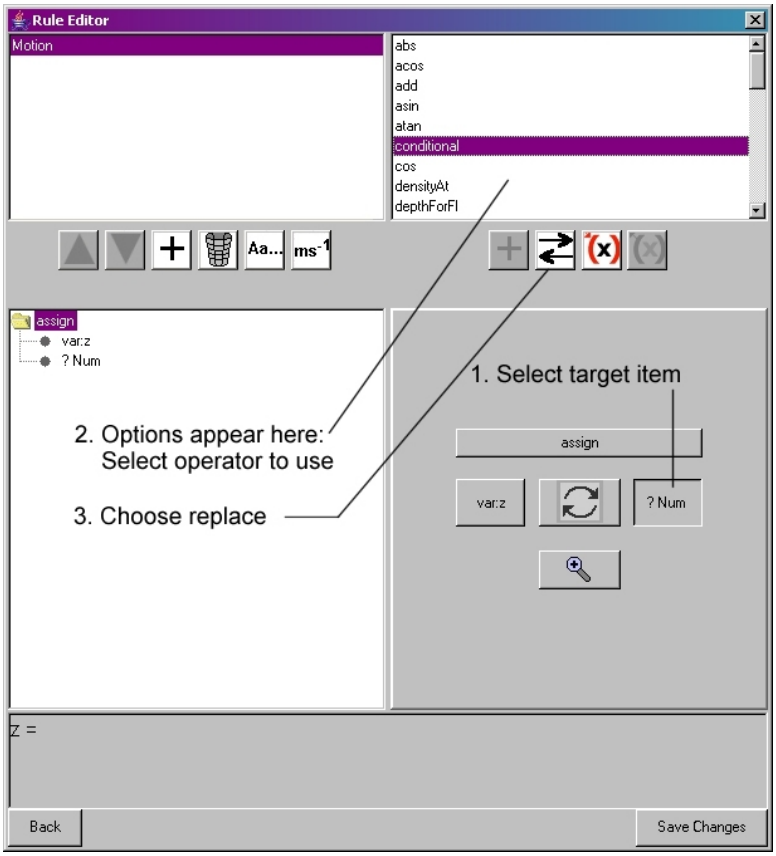

Figure A.7: Selecting a conditional function for the right-hand side.

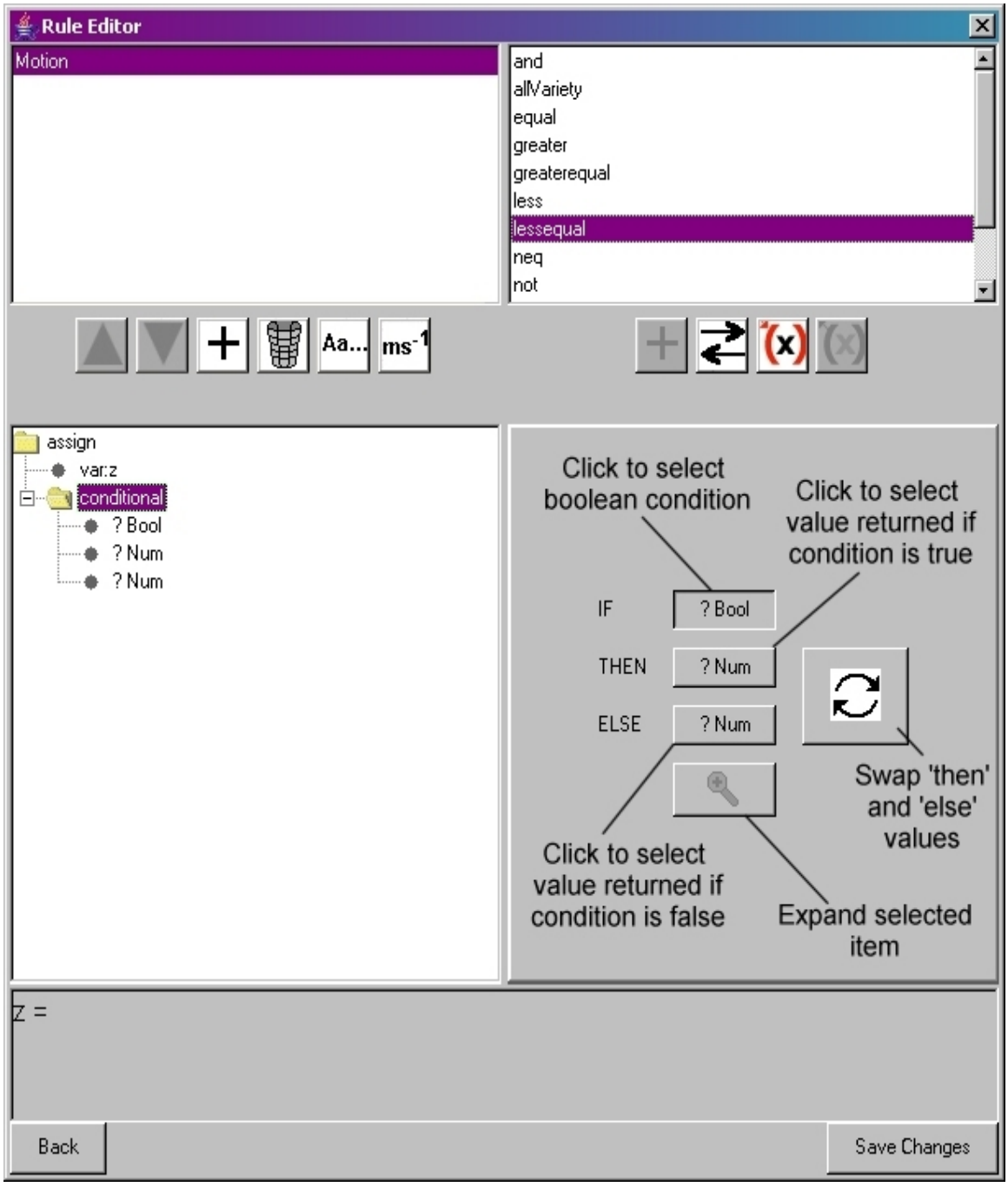

Figure A.8: Editing the conditional function.

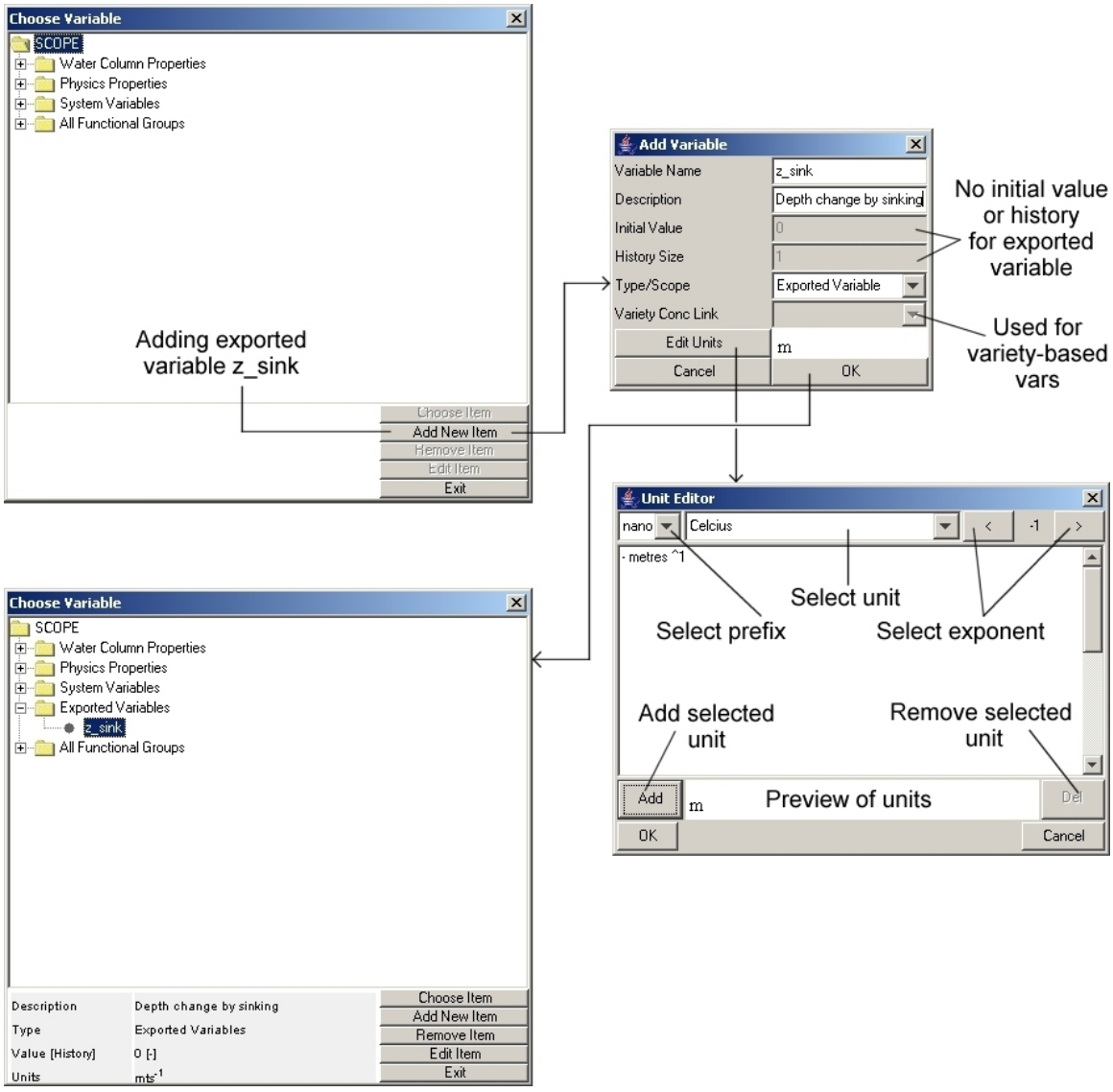

Figure A.9: Creating an exported variable

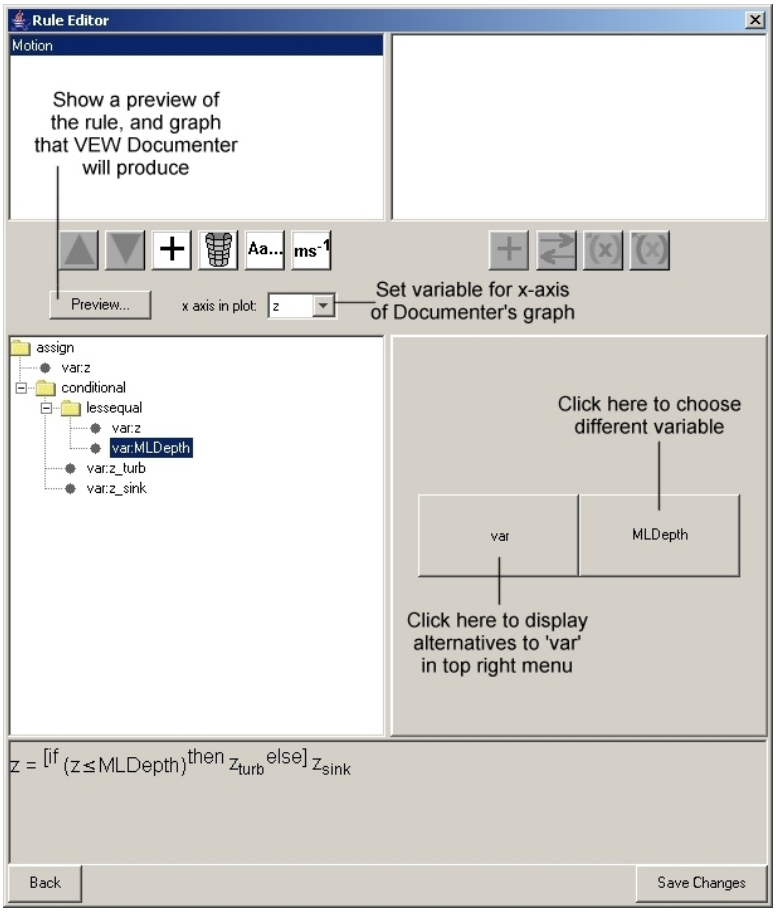

Figure A.10: The completed diatom motion equation

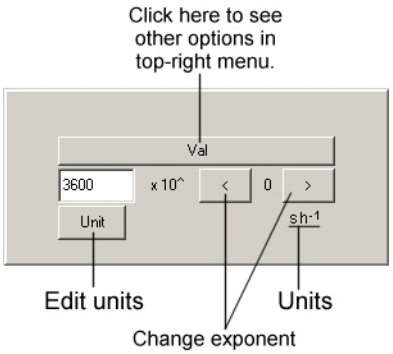

Figure A.11: Editing a numerical value

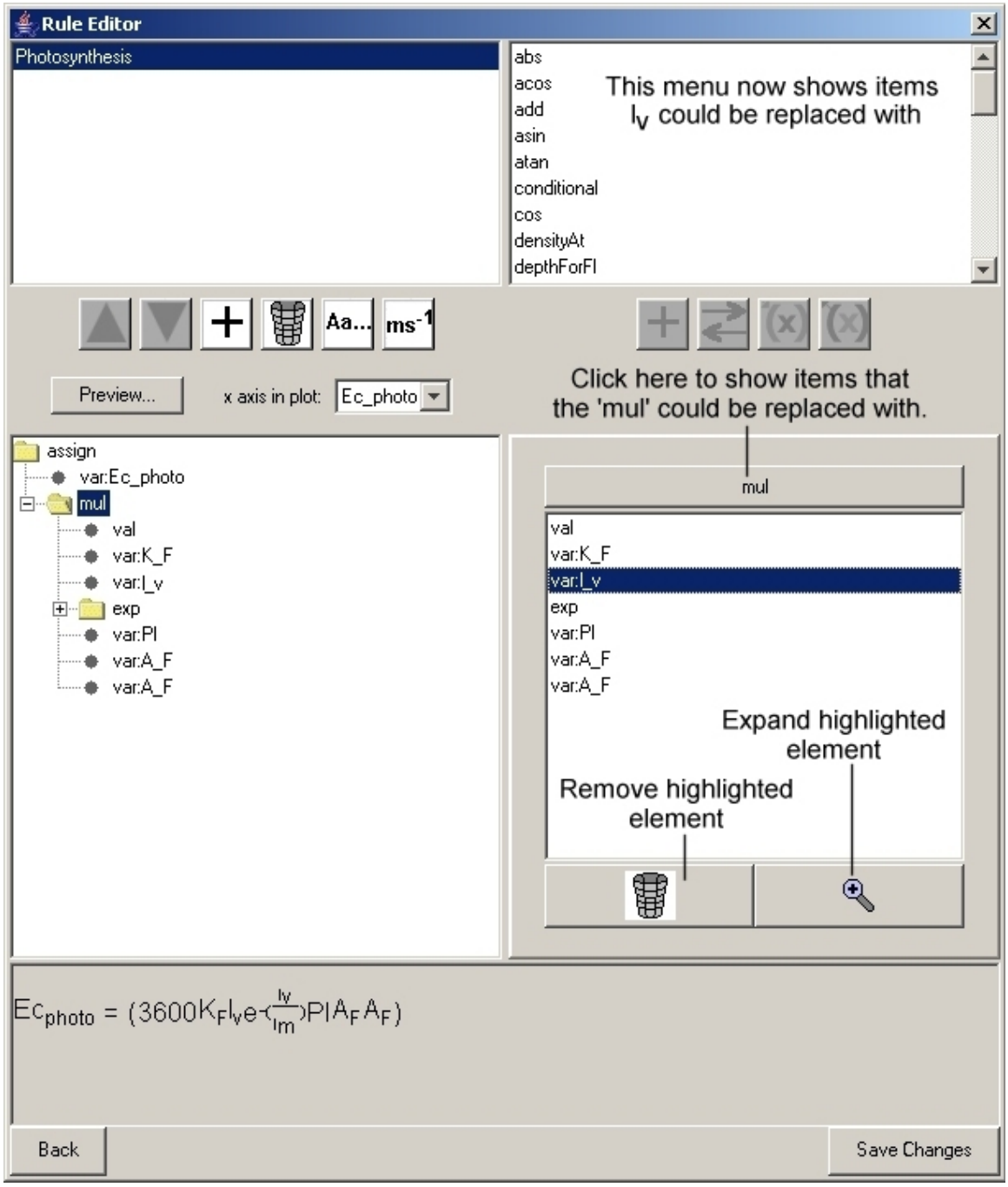

Figure A.12: Expressions with multiple arguments

Bibliography

[1] Jean-Marc Ame, Colette Rivault, and Jean-Louis Deneubourg. Cockroach aggregation based on strain odour recognition. *The Association for the Study of Animal Behaviour*, 68(4):793–801, August 2004.

[2] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[3] J.W. Baretta, W. Ebenhoh, and P. Ruardij. The european regional seas ecosystem model, a complex marine ecosystem model. *Neth. J. Sea Res.*, 33:233–246, 1995.

[4] Wolfgang Barkmann and John Woods. The lagrangian ensemble model, le'98 science modules. *Virtual Plankton Ecology Technical Report, Imperial College Department of Earth Science Engineering*, 6, July 1998.

[5] T. A. Barrell. A tool for modelling epidemiology in virtual plankton ecosystems. *Imperial College Department of Computing, M.Sc*, 1999.

[6] A.S. Berman. Daliworld: From info highway to digital ocean (http://www.usatoday.com/tech/2001-09-06-net-interest.htm). *USA Today*, 2001.

[7] Niall Broekhuizen. Simulating motile algae using a mixed eulerian-lagrangian ap-

proach: does motility promote dinoflagellate persistence or co-existence with diatoms? *Journal of Plankton Research*, 21(7):1191–1216, July 1999.

[8] Miguel de Arrese. Upward transport of chemicals by migrating zooplankton. *Dep. of Earth Sciences and Engineering, Imperial College, M.Eng Thesis*, May 2002.

[9] Alan Dorin. Building artificial life for play. *Artificial Life*, 10(1):99–112, 2004.

[10] Keith Downing. Euzone: Simulating the evolution of aqautic ecosystems. *Artificial Life*, 3(4):307–333, 1997.

[11] Christopher G. Langton (Editor). Artificial life: An overview. *Massachusetts Institute of Technology Press*, 1995.

[12] A. M. Edwards and A. Yool. The role of higher predation in plankton population dynamics. *Journal of Plankton Research*, 2(6):1085–1112, 2000.

[13] P.R. Epstein. Climate, ecology and human health. *Consequences*, 3(2), 1997.

[14] M.J.R. Fasham, H. W. Ducklow, and S. M. McKelvie. A nitrogen-based modelof plankton dynamics in the oceanic mixed layer. *Journal of Marine Research*, 48:591–639, 1990.

[15] International Institute for Applied Systems Analysis (IIASA www.iiasa.ac.at). Adise - adaptive dynamics integrated simulation environment. 2002.

[16] E. Gallopoulos, E.N. Houstis, and J.R. Rice. Future research directions in problem solving environments. *Technical Report CSD–TR–92–032, Department of Computer Sciences, Purdue University*, pages 92–132, 1992.

[17] Haskell. A purely functional language, (http://www.haskell.org/). December 2004.

[18] K. A. Hawick, C. J. Scogings, and H. A. James. Defensive spiral emergence in a predator-prey model. *Conference on Complex Systems, Cairns, Australia*, CSTN-010, 2004.

[19] Paul Houle. Rngpack, (http://www.honeylocust.com/rngpack/). 2003.

[20] Elias N. Houstis. The role of problem solving environments in engineering and mathematics education. *University of Thessaly Department of Computer and Communications Engineering*, 2003.

[21] P.T. Hraber, T. Jones, and S. Forrest. The ecology of echo. *Artificial Life*, 3(3):165–190, 1997.

[22] Exxon Valdese Oil Spill (http://www.evostc.state.ak.us/facts/qanda.html). Information page. 2005.

[23] Gameware Development (http://www.gamewaredevelopment.co.uk/creatures_index.php). Creature labs website. 2004.

[24] Swarm (http://www.swarm.org). Online information and software.

[25] R.G. Knox, V.L. Kalb, D.J. Kendig, and E.R. Levine. Workbench for interactive simulation of ecosystems (wise). *IEEE Computational Science and Engineering*, 4(3):52–60, 1997.

[26] Kremer and Nixon. A coastal marine ecosystem: Simulation and analysis. *Ecological Studies*, 24, 1978.

[27] C-C Liu and J.D. Woods. Prediction of ocean colour: Monte carlo simulation applied to a virtual ecosystem based on the lagrangian ensemble method. *International Journal of Remote Sensing*, 25(5):921–936, 2004.

[28] MathWorks. Mathematica, (http://www.wolfram.com/products/mathematica/). 2004.

[29] MathWorks. Matlab (http://matlab.mathworks.com/matlab.html). 2004.

[30] R.M. May. *Stability and complexity in model ecosystems.* 1973.

[31] Jacqueline Michell, Gary Shigenaka, and Rebecca Hoff. Oil spill response and clean-up technique (chapter 5). *Hazardous Materials Response and Assessment Division, National Oceanic and Atmospheric Administration*, 4(92).

[32] Gordon S. Novak, Jr. Conversion of units of measurement. *IEEE Transactions on Software Engineering*, 21(8):651–661, August 1995.

[33] D.J.B. Osborne. Particle size normalisation for virtual plankton ecology. *Imperial College Department of Computing, M.Sc project*, September 1999.

[34] Norman H. Packard and Mark A. Bedau. Artificial life (http://www.reed.edu/ mab/papers/ecs.pdf). *Encyclopedia of Cognitive Science, Macmillan (in press)*, 2000.

[35] J.K. Parrish and W.M. (editors) Hammer. Animal groups in three dimensions. *Cambridge University Press*, 1997.

[36] Lucas Partridge. A food web model based on the lagrangian ensemble method. *Unpublished report, University of Southamptonn*, 2000.

[37] Pendar Paykari, Ali Kamkarfar, Raj Bhattacharjee, Dipesh Patel, Wes Hinsley, and Tony Field. Livesim. *Imperial College Department of Computing, M.Eng. Group Project*, December 2004.

[38] Alfredo Pina, Francisco J. Seron, and Diego Gutierrez. The alvw system: an interface for smart behavior-based 3d computer animation. *2nd International Symposium on smart graphics, Hawthorne, New York*, pages 17–20, 2002.

[39] Cathryn J Polinsky. Uses of boids and similar flocking algorithms in movies (http://www.cs.swarthmore.edu/ polinsky/cs97/movies.html). 2004.

[40] E.E. Popova, M.J.R. Fasham, A.V. Osipov, and V.A. Ryabchenko. Chaotic behaviour of an ocean ecosystem model under seasonal external forcing. *Journal of Plankton Research*, 19(10):1495–1515, 1997.

[41] Craig W. Reynolds. Boids, simulated flocking. *(http://www.red3d.com/cwr/boids/)*, 2004.

[42] Arun Rishi. Vew controller (species and varieties). *Imperial College Department of Computing, M.Eng Report*, June 2003.

[43] Adrian Rogers. Vew analyser. *Imperial College Department of Computing, M.Eng Project*, June 2003.

[44] Adrian Rogers. A virtual ecology workbench for creating three-dimensional virtual plankton ecosystems. *Imperial College Department of Earth Science Engineering, Ph.D Transfer Report*, 2005.

[45] O. Rud and M. Kahru. Monitoring of harmful algal blooms in the baltic sea. *Remote Sensing*, 26:8–10, 1995.

[46] Matteo Sinerchia. Testing theories of fisheries recruitment. *Imperial College Department of Earth Science Engineering, Ph.D Transfer Report*, 2005.

[47] Simon Smith. Modelling the properties of copepod digestion. *Unpublished Technical Report, Imperial College Department of Earth Science Engineering*, 2000.

[48] J.H. Steele and E.W. Henderson. The role of predation in plankton models. *Journal of Plankton Research*, 14(5):157–172, 1992.

[49] D.J.T. Sumpter and S.J. Martin. The dynamics of virus epidemics in varroa-infested honey bee colonies. *Journal of Animal Ecology*, 73:51–63, 2004.

[50] Tal Zvi Trachtman. Destiny studio, rule-based agent society modelling and simulation environment for non-programmers. *Imperial College Department of Computing, M.Eng Project*, June 2004.

[51] Northwestern University. Netlogo (http://ccl.northwestern.edu/netlogo/). *The Center for Connected Learning and Computer-Based modelling*, 2004.

[52] Silvana Vallerga. Discussions,. *Imperial College Department of Earth Science Engineering*, 2004.

[53] Evan Weaver. Vew documenter. *Unpublished documentation, Imperial College Department of Earth Sciences and Engineering*, 2004.

[54] J. D. Woods. Simulating the plankton ecosystem by the lagrangian ensemble method. *Philosophical Transactions Royal Society, London*, B 343:27–31, 1987.

[55] J. D. Woods. The virtual ecology workbench. *Imperial College Department of Earth Sciences and Engineering, Virtual Plankton Ecology Technical Report 2*, 2000.

[56] J. D. Woods. The lagrangian ensemble metamodel for simulation plankton ecosystems. *Progress in Oceanography (submitted)*, March 2005.

[57] J. D. Woods and R. Onken. Diurnal variatoin and primary production in the ocean - preliminary results of a lagrangian ensemble model. *Journal of Plankton Research*, 4:735–756, 1982.

[58] John Woods, Tony Field, Adrian Rogers, Matteo Sinerchia, and Wes Hinsley. Discussions regarding top and bottom closure. *Imperial College Department of Earth Science Engineering*, December 2004.

[59] John Woods, Angelo Perilli, and Wolfgang Barkmann. Stability and predictability of a virtual plankton ecosystem created with an individual-based model (to be published). *Progress in Oceanography (in press)*, 2005.