Process-Based Models of Dynamical Systems: Representation and Induction

Darko Čerepnalkoski

Doctoral Dissertation
Jožef Stefan International Postgraduate School
Ljubljana, Slovenia, September 2013

MEDNARODNA PODIPLOMSKA ŠOLA JOŽEFA STEFANA
JOŽEF STEFAN INTERNATIONAL POSTGRADUATE SCHOOL
Ljubljana, Slovenia

Darko Čerepnalkoski

Process-Based Models of Dynamical Systems: Representation and Induction

Doctoral Dissertation

Predstavljanje in učenje procesnih modelov dinamičnih sistemov

Doktorska disertacija

Supervisor: Prof. Dr. Sašo Džeroski

Co-Supervisor: Assoc. Prof. Dr. Ljupčo Todorovski

Ljubljana, Slovenia, September 2013

Index

Abstract

Mathematical models are the principal tool for comprehending real-world systems. Mathematical models, not unlike the systems they aim to represent, come in many different flavors. Models provide an insight in the inner workings of the system, revealing the processes that govern the behavior of the system. They can be used to recreate the system behavior or simulate the behavior of the system under hypothetical conditions that have not yet been observed. Models are also used to extrapolate the current behavior, thus making a prediction of the future behavior of the system.

Process-based modeling is a modeling technique that uses two-level approach for modeling dynamical systems. It models systems on a purely qualitative level in terms of entities and processes that involve those entities. On a quantitative level, all entities and processes are given a quantitative formulation which is then translated into a set of ordinary differential equations.

Our aim is to develop a novel approach to inductive process modeling, including a novel formalism for representing such models and a platform for learning such models from data and domain knowledge. The approach should address current challenges to inductive process modeling, grouped in two major categories. The first category addresses the representation formalism and directly influences the amount of background knowledge that can be incorporated as well as the intelligibility and ease of use by domain experts. The second category addresses the variety of modeling scenarios that can be tackled and includes the observability scenarios and objective functions that can be used. It also addresses the efficiency of the parameter estimation subsystem.

In this dissertation, we present ProBMoT, a tool for automated modeling of dynamical systems that addresses both structure identification and parameter estimation. It takes into account domain knowledge formalized as templates for the components of the process-based models: entities and processes. Taking a conceptual model of the system, the library of domain knowledge, and measurements of a particular dynamical system, it identifies both the structure and numerical parameters of the appropriate process-based model. ProBMoT has two main components corresponding to the two subtasks of modeling. The first component is responsible for generating candidate model structures that adhere to the conceptual model specified as input. The second subsystem uses the measured data to find suitable values for the constant parameters of a given model by using parameter estimation methods. ProBMoT uses model error to rank model structures and select the one that fits measured data best.

We investigate the influence of the selection of the parameter estimation methods on the structure identification. We consider one local (derivative-based) and one global (metaheuristic) parameter estimation method. As opposed to other comparative studies of parameter estimation methods that focus on identifying parameters of a single model structure, we compare the parameter estimation methods in the context of repetitive parameter estimation for a number of candidate model structures. The results confirm the superiority of the global optimization methods over the local ones in the context of structure identification.

Povzetek

Matematični modeli nam pomagajo pri razumevanju naravnih sistemov. Izbran sistem lahko opišemo z več različnimi modeli. Z njihovo pomočjo lahko opazujemo notranje delovanje poljubnega sistema, pri čemer odkrivamo procese, ki vplivajo na njegovo obnašanje. Z omenjenimi modeli lahko posnemamo delovanje sistema ali pa simuliramo njegovo obnašanje pod različnimi pogoji. Modele uporabljamo tudi za napovedovanje obnašanja sistema v prihodnosti.

Procesno modeliranje predstavlja poseben pristop k modeliranju, ki za opis poljubnega dinamičnega sistema uporablja dve ravni. Na kvalitativni ravni se za opis sistema uporabljajo entitete in procesi, ki te entitete povezujejo, na kvantitativni ravni pa so vsem entitetam in procesom dodeljeni kvantitativni opisi, ki se v postopku modeliranja prevedejo v sistem navadnih diferencialnih enačb.

Naš cilj je razviti nov pristop k induktivnemu procesnemu modeliranju, ki temelji na novem formalizmu za opis modelov in računalniškem okolju, namenjenem učenju modelov na osnovi podatkov in domenskega znanja. Predlagan pristop se sooča s trenutnimi izzivi na področju induktivnega procesnega modeliranja, ki jih lahko razdelimo v dve večji skupini. Prva obravnava formalizem za zapis domenskega znanja. Ta neposredno vpliva na količino znanja, ki ga lahko vključimo v postopek modeliranja, pri čemer sta pomembni predvsem razumljivost formalizma in njegova uporabnost. Druga skupina obravnava vrsto različnih scenarijev in objektivnih funkcij, ki jih lahko uporabimo v postopku modeliranja. Hkrati obravnava učinkovitost podsistema za umerjanje parametrov.

V doktorski disertaciji predstavljamo ProBMoT, orodje za avtomatizirano modeliranje dinamičnih sistemov, ki obravnava tako odkrivanje strukture kot tudi umerjanje parametrov modela. V ta namen uporablja domensko znanje, zapisano v obliki predlog komponent procesnih modelov, natančneje entitet in procesov. ProBMoT išče primerno strukturo in ustrezne vrednosti parametrov procesnega modela na osnovi konceptualnega modela opazovanega dinamičnega sistema, knjižnice domenskega znanja in meritev. Orodje sestavljata dve osnovni komponenti, ki ustrezata dvema fazama modeliranja. Prva komponenta je odgovorna za generiranje različnih kandidatnih struktur, ki sledijo opisu sistema, podanemu v konceptualnem modelu. Druga komponenta uporablja merjene podatke za iskanje primernih vrednosti parametrov modela, pri čemer se poslužuje posebnih metod umerjanja. Kot merilo za razvrstitev generiranih struktur uporablja ProBMoT napako modela, na osnovi katere izbere tudi strukturo, ki zagotavlja najboljše ujemanje rezultatov modela z meritvami.

V disertaciji preiskujemo vpliv izbrane metode za umerjanje parametrov na proces odkrivanja strukture modela. V ta namen obravnavamo dve metodi za umerjanje parametrov: lokalno, ki temelji na odvodih funkcije, in globalno (metahevristično). V primerjavi z predhodnimi raziskavami, ki so preučevale metode za umerjanje parametrov z vidika iskanja primernih vrednosti parametrov ene same strukture modela, v naši disertaciji primerjamo metode za umerjanje v kontekstu ponavljajočega se umerjanja več različnih kandidatnih struktur. Naši rezultati potrjujejo, da je tudi z vidika umerjanja več različnih generiranih struktur modela globalna optimizacijska metoda boljša od lokalne.

1 Introduction

A scientific model is a description of a system using a formal language. Modeling is the process in which such scientific models are created. By using models, we can conceptualize systems from the real world so that we can analyze and better understand them (Box and Draper, 1987).

Mathematical models are the principal tool for comprehending real-world systems. Mathematical models, not unlike the systems they aim to represent, come in many different flavors (Gershenfeld, 1998). Models provide an insight in the inner workings of the system, revealing the processes that govern the behavior of the system. They can be used to recreate the system behavior or simulate the behavior of the system under hypothetical conditions that have not yet been observed. Models are also used to extrapolate the current behavior, thus making a prediction of the future behavior of the system.

The main task of modeling is to establish a model for an observed system. This includes identification of the structure of the model and estimation of its numerical parameters. Of a particular interest to science and engineering is a class of systems referred to as dynamical systems (Luenberger, 1979).

There are different definitions of dynamical systems in the literature. They all agree that dynamical systems are systems that change their state over time. In the context of this dissertation, we present the following definition by Aoki and Hiraide (1994).

Dynamical systems are characterized by three main components:

- *time* T, which is an ordered set.
- *state space* S, which is the set of all possible states that the system can assume. Any given state provides a complete description of the system.
- *evolution rule* R, that describes the transition from one state to another at a given point in time. We can represent it in the form of a function $R : T \times S \to S$

Dynamical systems can be categorized into several types, depending on the type of their components. The time of a dynamical system can be either discrete or continuous. The state space can also be discrete or continuous, whereas the evolution rule can have either a deterministic or a stochastic form. This results in eight different types of dynamical systems, some more common than the others. We concentrate our attention on one type of dynamical systems—those characterized by continuous time, a continuous state space and a deterministic evolution rule.

This class of dynamical systems is especially common in natural sciences and engineering and describes most of the tangible dynamical systems that appear in practice. A common choice for modeling such systems are Ordinary Differential Equations (ODEs). Differential equations (Braun, 1993) are a natural choice because they deal with continuous time and space and provide explanations that are time invariant. ODEs can be used to simulate the system's behavior and compare the simulation to empirically obtained data. In addition, ODEs can be used to make numerical predictions about the future states that the system will assume.

One drawback of using ODEs, however, is that the more complex the system gets, the longer the equations tend to grow, which makes them extremely difficult to comprehend. It gets increasingly difficult to relate parts of an equation to phenomena of the system and to track the genealogy of the terms of the equation with respect to the processes that occur in the system. A much more desirable representation would encapsulate the specific terms from the equations into meaningful concepts that the domain expert can relate to and grasp with relative ease.

When exploring a system, scientists also ponder the qualitative aspects of the system, trying to decompose it into simpler, more manageable concepts and analyze the relationships between those concepts. A formalism for modeling dynamical systems should thus include facilities that would enable constructing the system from smaller, more manageable pieces and describing the system in terms of the relationships of its components. A key property of such a formalism would be the ability to translate a model to a set of ODEs, so that it can be worked with quantitatively and be brought to the level of detail and precision which is common for dynamical systems.

1.1 Modeling Dynamical Systems and Artificial Intelligence

The modeling of dynamical systems involves several stages (Luenberger, 1979). When devising a model of a dynamical system, the first task is to decide on the structure of the model. Starting with a conceptual model of the system of interest and using knowledge about modeling the specific domain, the modeling expert selects the most suitable model structure for the observed ecosystem. This is the structure identification phase of the modeling process.

Once the model structure has been specified, the modeler has to determine suitable values for the numerical parameters of the model. To this end, he can use different techniques for estimating the values of the parameters from measured data about the behavior of the system. The data include measurements of the factors that drive the changes of the system, as well as the quantities that determine the state of the system. This is the parameter estimation phase of the modeling process.

The task of modeling dynamical systems has been addressed extensively within several research areas. In electrical engineering, the area of system identification is concerned with finding models using measured data. In artificial intelligence, the area of qualitative reasoning is concerned with formulating understandable models of dynamic systems, while the area of equation discovery (a topic in machine learning) tries to learn both model structure and parameters from data.

The work in this thesis is situated at the intersection of the above areas. In the following subsections, we briefly discuss approaches from each area. System identification is discussed first, followed by evolutionary computing, equation discovery and qualitative reasoning. In addition, the topics of numerical simulation of ODEs and nonlinear optimization are also introduced, since the work presented in the thesis employs existing methods developed within the corresponding areas.

1.1.1 System Identification

System identification (Ljung, 1999) focuses on identifying a suitable model for a given systems based on observed and measured data of the system behavior. Identifying the model involves two tasks: structure identification and parameter estimation. The most widely used system identification approaches focus on the parameter estimation task. The structure of the model is typically assumed to be known or given by the domain expert. When the structure of the model is not known, it is chosen form some well-defined class of model structures such as linear equations, polynomials, neural networks, or regression trees.

Classical techniques used in system identification like linear regression, finite impulse response, or autoregressive integrated moving average are very suitable for modeling linear dynamical systems. When the system is known to be nonlinear, it is common to resort to linearizing the system before using the aforementioned techniques. Nonlinear system identification (Nelles, 2001) fares much better at this challenging task. Nonlinear modeling techniques, including neural networks, fuzzy, and neuro-fuzzy models, have been used used with great success to model various industrial systems.

Classical, as well as nonlinear, system identification techniques suffer from the same shortcomings. They do not necessarily reveal the processes that govern the behavior of the observed system. The models are primarily black-box models, which can be very successful in predicting future behavior, but dot not disclose the internal structure of the systems and do not provide the domain expert with an insight into why the systems behaves in a certain manner.

1.1.2 Evolutionary Computing

Evolutionary computing (Eiben and Smith, 2003) is the collective name for a vast range of computational methods based on principles of biological evolution, such as natural selection and genetic inheritance. These methods can be applied to problems that involve both continuous and combinatorial optimization.

Genetic programming (Koza, 1992, 1994) is a evolutionary technique for the automatic generation of computer programs residing on concepts from the theory of evolution. Genetic programming can be used to evolve an algebraic expression as part of an equation representing measured input-output response data. The technique has been applied successfully for the identification of nonlinear structures of dynamical models from experimental data (Gray et al., 1998). More recently, Schmidt and Lipson (2009) have demonstrated the discovery of physical laws directly from experimentally captured data with the use of genetic programming.

The crucial drawback of genetic programming in the context of modeling dynamical systems is the limited amount of background knowledge that can be used in the generation of candidate model structures. Genetic programming typically relies on a pool of mathematical functions and operators that are combined to obtain the resulting model structure. The best-fitting model may often contain implausible combinations of otherwise allowed mathematical functions. This limits the applicability of genetic programming in domains where theoretical modeling knowledge is abundant and constrains the space of plausible models.

1.1.3 Equation Discovery

Machine learning (Mitchell, 1997) is largely concerned with the extraction of new knowledge from accumulated experience presented in the form of training examples. Equation discovery (Langley et al., 1987) is a subfield of machine learning focusing on developing methods for discovery of quantitative laws, expressed as equations, from measured data.

Equation discovery methods can be used to learn an equation based model of a system from measurements and observations. Thus, they are closely related to system identification. Unlike system identification, however, equation discovery methods simultaneously tackle both structure identification and parameter estimation. Moreover, equation discovery methods very commonly rely on system identification techniques for solving the parameter estimation task.

A number of methods have been developed within equation discovery, including BACON (Langley et al., 1987), ABACUS (Falkenhainer and Michalski, 1986), COPER (Kokar, 1986), EF (Zembowicz and Żytkow, 1992), E* (Schaffer, 1993), LAGRANGE (Džeroski and Todorovski, 1995), and SDS (Washio and Motoda, 1997). They use a predefined, reasonably small class of possible equations structures, such as polynomials or trigonometric functions. They allow some degree of user influence in the form of parameters of the algorithm like the degree of the induced polynomial or the number of terms in the equation. The background knowledge used by these early algorithms is very limited, and usually comes in the form of information about the measurement units of the state variables.

These early equation discovery methods mostly focus on discovering algebraic equations, and are not suitable for modeling dynamical systems. LAGRANGE, however, can discover a set of differential equations by computing beforehand the numerical derivatives of the observed state variables. Unfortunately, numerical differentiation introduces large numerical errors and is very sensitive to noisy data. GOLDHORN (Križman et al., 1995) uses numerical integration instead of differentiation, thus improving the numerical stability of the algorithm.

The equation discovery method LAGRAMGE (Todorovski and Džeroski, 1997) is capable of discovering ordinary differential equations whose right-hand side can be derived using a user provided context-free grammar. Each equation structure considered during the search contains one or more generic constant parameters. The parameters are estimated using the parameter estimation technique ALG-717 by Bunch et al. (1993).

1.1.4 Qualitative Reasoning

Within artificial intelligence (Russell and Norvig, 1995), the filed of qualitative reasoning (Kuipers, 1994) deals with qualitative representations of physical systems and reasoning about them. The most commonly used formalism for specifying qualitative models are qualitative differential equations (QDEs), simulated with the QSIM algorithm (Kuipers, 1994).

Qualitative process theory (Forbus, 1984) provides a framework for building qualitative models. The framework relies on the notion of a physical process, which can apply to some components of the system and influence their parameters. The model building tool QPC (Crawford et al., 1990) takes the approach of qualitative process theory and organizes domain-specific background knowledge in libraries of model fragments that specify models of processes or components of the observed system.

The PRET reasoning system (Bradley et al., 2001) aims to automate modeling by building a layer of artificial intelligence techniques around a set of traditional formal engineering methods. PRET takes a generate-and-test approach, using a meta-domain theory that tailors the space of candidate models. It then tests these models against the behavior of the observed system using more-general mathematical rules.

Garp3 (Bredeweg et al., 2009) is a user-oriented workbench for building, simulating, and inspecting qualitative models. It uses a graphical user interface, where models are presented as graphs and each node is a system quantity or a process. It is suitable for creating conceptual models in situations where numerical information is sparse or unavailable.

1.1.5 Numerical Simulation of ODEs

The methodology presented in this dissertation relies on well-established methods for numerical simulation of ODEs. In this section we briefly introduce the most influential methods to solving ODEs and their most widely spread software implementations.

The simplest method for numerical simulation of ODEs is the Euler's method (Press et al., 2007) which is based on the geometric interpretation of the derivative as the slope of the tangent line touching the curve that represents the true value of the state variable. The Euler's method is numerically unstable and suffers from and introduces a global simulation error proportional to the step size which makes it unusable on anything except the simplest differential equations.

An important family of methods for solving ODEs are Runge-Kutta methods (Verner, 1978, 1979). Runge-Kutta (RK) methods are linear one-step methods that use a series of internal approximations (stages) to reduce the local, and hence the global simulation error. They are not very efficient because they require multiple functional evaluations per step in order to produce a simulation.

The most widely spread class of methods for solving ODEs are *linear multistep methods* (Butcher, 2003). Linear multistep (LM) methods improve efficiency over RK methods by reusing the computation from previous steps rather than discarding it. A k-step linear multistep method uses a linear combination of the k previous points and derivative values to compute the current value. Although there are many sets of LM methods derived, there are two families which are most commonly used: Adams-Moulton methods and the backward differentiation formulas.

Adams-Moulton formulas (Sampine and Gordon, 1975; Gear, 1971) are implicit, and hence a set of nonlinear equations must be solved in each step. Typically, simple functional iterations are used to solve the nonlinear equations. Adams-Moulton methods are very efficient and well suited to applications that require the values at a large number of points, but are limited to nonstiff differential equations.

The most commonly used multistep methods for solving sets of stiff equations are the backward differentiation formulas (BDFs) (Ascher and Petzold, 1998; Iserles, 1996). Similarly to Adams-Moulton methods, BDFs are also implicit formulas and require solving nonlinear equations at each step, but in contrast to them, BDFs require some form of Newton iterations. The Newton iterations nonlinear solver require the solution of linear systems involving the Jacobian of the system.

There is a wide array of FORTRAN solvers, among which LSODE (Livermore Solver for Ordinary Differential Equations) and VODE (Variable-coefficient ODE solver) have been heavily utilized.

The package LSODE (Hindmarsh, 1980, 1983) has been designed for the numerical solution of a system of first-order ordinary differential equations given the initial values. It includes a variable-step, variable-order Adams-Moulton method (suitable for nonstiff problems) of orders 1 to 12 and a variable-step, variable- order backward differentiation formula method (suitable for stiff problems) of orders 1 to 5. LSODE allows for a manual selection of method order in addition to the default automatic selection.

VODE (Brown et al., 1989) is a general purpose ODES solver very similar to LSODE. The main difference between VODE and LSODE is that VODE uses variable-coefficient methods (fixed-leading coefficient form) instead of the fixed-step-interpolate methods in LSODE. Brown et al. (1989) have shown that VODE is often more efficient than LSODE.

The library CVODE (Cohen and Hindmarsh, 1996; Hindmarsh et al., 2005) is a reimplementation of the VODE package in the C programming language. It contains all of the methods present in VODE, with some considerable improvements in the organization of the package. CVODE implements a variable-order, variable-step multistep methods including both Adams-Moulton and BDFs.

1.1.6 Parameter Estimation Through Nonlinear Continuous Optimization

Classical approaches to nonlinear continuous optimization are mainly local optimization methods (such as direct-search and derivative-based methods) that rapidly converge to the optimum, provided that the search is started from an initial point that is in close proximity of the optimum. As these methods do not have mechanism to escape from the local optima, they only guarantee local convergence. Derivative-based methods are an adequate choice for smooth and unimodal objective functions, but they can fail if the landscape is discontinuous, non-smooth, multi-modal or ill-conditioned. The disadvantage of the direct-search method is that they become less efficient for high-dimensional problems. Therefore, it is recommended to use global optimization approaches that are more robust regarding the dimensionality and the landscape characteristic of the search space.

Global optimization approaches can be divided into deterministic (exact) and stochastic (probabilistic). The deterministic methods (e.g., branch and bound, interior-point, cutting planes etc.) can locate the global optima and assure their optimality, but do not guarantee that they can solve any type of global optimization problems in finite time. Stochastic methods, on the other hand, rely on probabilistic search rules to find good solutions (Törn et al., 1999). They can locate the neighborhood of the global optima relatively quickly, but their efficiency comes at the cost of not being able to guarantee global optimality. In the last two decades, special attention has been given to meta-heuristics: These are general-purpose algorithms that can find acceptable solutions in a reasonable time-frame in complex and large search domain. Most meta-heuristics are inspired by natural processes such as evolution (evolutionary algorithms) or social behavior of biological organisms, e.g., ant colony optimization (Dorigo and Stützle, 2004).

Another notable approach is Covariance Matrix Adaptation - Evolution Strategy (CMA-ES) developed by Hansen and Ostermeier (1996). This method uses evolutionary strategy to update the covariance matrix of this distribution, hence learning a second order model of the underlying objective function.

The methodology developed in the dissertation utilizes three optimization methods. One of them is a local method, Alogorithm 717—an efficient and state-of-the-art implementation of the classical derivative-based approach to local optimization.The other two are global optimization methods. Recently, many methods for nonlinear optimization have been introduced and compared in the context of standardized black-box optimization benchmarks (Hansen et al., 2010). In the thesis, we selected two of them. One is Differential Evolution, a classical evolutionary method for nonlinear optimization. The other global method is Differential Ant-Stigmergy Algorithm, a contemporary meta-heuristic optimization method.

Algorithm 717 (ALG-717)

Algorithm 717 (ALG-717) is a set of modules for solving the parameter estimation problem in nonlinear regression models, including the nonlinear least-squares, maximum likelihood, and some robust fitting problems (Bunch et al., 1993). The basic method is a generalization of NL2SOL - an adaptive nonlinear least-squares algorithm, which uses a model/trust-region technique for computing trial steps along with an adaptive choice for the Hessian model. Since ALG-717 is not a global search algorithm, we wrapped the original procedure in a loop of restarts with randomly chosen initial points, providing in some way a simple global search. The number of restarts was set to use a number of function evaluations comparable to that of the other method (DASA). We used the module for constraint (on parameter bounds) optimization with user-supplied routines for the first and second-order derivatives of the objective function.

The Differential Ant-Stigmergy Algorithm (DASA)

The DASA algorithm was proposed by (Korošec et al., 2012). It is a version of an Ant Colony Optimization (ACO) meta-heuristic, designed to successfully cope with high-dimensional continuous optimization problems. The rationale behind the algorithm is in memorizing the move in the search space that improves the current best solution, and using it in further searches. The algorithm uses pheromones as a means of communication between ants (a phenomenon called stigmergy), combined with graph representation of the search space.

The most important property of DASA is that it transforms the problem into a graph-search problem by fine-grained discretization of the continuous domain of the parameters' differences, unlike the common way of discretizing parameters values. The parameters' differences assigned to the graph vertices are used to navigate through the search space.

Differential Evolution

Differential evolution (DE) is a simple and efficient population-based heuristic for optimizing real-valued multi-modal functions, introduced by Storn and Price (1997, 1995). It belongs to the class of evolutionary algorithms based on the idea of simulating the natural evolution of a population P of individuals (candidate solutions) via the processes of selection, mutation and crossover.

The main difference between traditional evolutionary algorithms and DE is in the reproduction step, where for every candidate solution an offspring is created using a simple arithmetic (differential) mutation operation over three (or more) parents. Moreover, uniform crossover is introduced, in order to increase the diversity of the mutated solution. Finally, the offspring is evaluated and if its fitness (objective function) is better, it replaces the corresponding candidate solution in the population.

1.2 Inductive Process Modeling

Inductive process modeling (IPM) lies at the intersection of system identification, equation discovery and qualitative reasoning. It involves constructing a process-based model of a dynamical system from observed data about the system (capturing the behavior of the system over time). Process-based domain knowledge is also used in IPM, in addition to the observed data. The proponents of inductive process modeling claim that casting the domain and learned knowledge in terms of processes with associated equations is desirable for scientific and engineering domains, where such notations are commonly used.

In the following subsections be briefly summarize the existing approaches to inductive process modeling and their and reveal their limitations. They are addressed in much greater detail in Chapter 2.

1.2.1 State of the Art in IPM

LAGRAMGE 2.0 (Todorovski, 2003; Todorovski and Džeroski, 2006) is a modeling framework that integrates the knowledge-based (theoretical) approach to modeling with data-driven (empirical) modeling. The framework allows for integration of modeling knowledge specific to the domain of interest in the process of model induction from measured data. The knowledge is organized around the central notion of basic process in the domain and it includes models thereof as well as guidelines for combining models of individual processes into a model of the entire observed system. This framework uses the equation discovery tool LAGRAMGE (Todorovski and Džeroski, 1997) to heuristically search through the space of candidate models, match them against data, and find the one that fits the data best.

The background knowledge in IPM (Bridewell et al., 2008) is specified in terms of generic processes. They do not commit to particular variables or parameter values, but they indicate constraints on them, like the type of the variables and the range of the parameters. The background knowledge also contains a hierarchy of variables that descend from the same base type *number*.

HIPM (Todorovski et al., 2005) extends IPM's formalism and organizes process knowledge in a hierarchical manner. Hierarchical processes characterize an observed system's behavior at distinct levels. The hierarchical structure lets the induction method carry out search through an AND/OR space rather than an OR space, thus reducing the number of candidate models considered and ensuring that these models will make sense to domain scientists. This approach requires greater effort to encode background knowledge, but offsets this with a more efficient search over a space of more plausible models.

1.2.2 Limitations of Existing IPM Approaches

Our aim is to develop a novel approach to inductive process modeling, including a novel formalism for representing such models and a platform for learning such models from data and domain knowledge. The approach should address current challenges to IPM, grouped in two major categories. The first category addresses the representation formalism and directly influences the amount of background knowledge that can be incorporated as well as the intelligibility and ease of use by domain experts. The second category addresses the variety of modeling scenarios that can be tackled and includes the observability scenarios and objective functions that can be used. It also addresses the efficiency of the parameter estimation subsystem.

Representation

LAGRAMGE 2.0 uses a declarative language for textually representing the background knowledge and models. The background knowledge for a domain of interest is specified in a domain-specific library. This library defines the components for building the models, as well as, the search space of possible model structures. If the domain expert has some beforehand knowledge of the specific system, he has to tailor the library to his specific scenario. This, in essence, means that each IPM task requires its own custom-build library.

IPM is unable to exclude combinations of processes when considering candidate model structures. Because the background knowledge typically contains many alternatives of the same conceptual process, IPM takes into consideration models which contain duplicate conceptual processes. IPM is also unable to imply that certain processes may only appear in conjunction with some other particular process or not appear at all. This issue arises in IPM's treatment of nested processes.

HIPM does not introduce a modeling language for the representation of models and background knowledge. Instead, the models and background knowledge are expressed in its internal representation, directly in Python. While Python is a language with very little syntactic overhead and manages to capture the semantic relations of process-based modeling in a fairly clean manner, it is nevertheless a programming language. Having in mind that IPM tools are intended to be used not only by computer scientists, but even more so by domain experts from diverse fields including environmental sciences, computational biology, and earth sciences among others, a conventional programming language is not the optimal solution.

As the system complexity increases, so does the complexity of the model of the system. Large and complex systems are commonly represented through multi-compartment models. Disciplines like integrated environmental modeling (Laniak et al., 2013) or systems biology (Kitano, 2001), which focus on large-scale models employ compartments to structure and organize the models. Each compartment acts like a small-scale model focused on one segment of the system. Compartments can communicate and exchange information through well defined inter-compartmental processes. None of LAGRAMGE 2.0, IPM, and HIPM includes facilities to support multi-compartmental modeling.

The power of a formalism for representing models and background knowledge comes from its flexibility. A flexible formalism can support domain expert in formulating complex modeling scenarios. Advanced modeling facilities like compartments, hierarchies of modeling components, and incomplete models increase the flexibility a formalism and enable the modeling expert to perform sophisticated, well-targeted experiments. These facilities, however, also increase the complexity of the formalism. Defining the capabilities of the formalism, the relations and properties that hold between its object requires a rigorous mathematical specification. The existing methodologies LAGRAMGE 2.0, IPM, and HIPM do not contain mathematical specifications of their components.

Modeling Scenarios

Identifying the system structure and parameters becomes even more challenging when the system cannot be fully observed. The full description of the system in any point in time is contained in its state variables. When the number of state variables and the measurement technology permit, we can directly observe all state variables. Such a system is a fully observed system. In many real-world scenarios, however, we can only observe some of the state variables, whereas the rest of them are unobservable (or hidden). Such a system is a partially observed system. LAGRAMGE 2.0, IPM, and HIPM support the specification of partially observed systems. However, the modeling scenario may require that the quantities that are observed can even be a function of several state variables. Handling such versatile observational scenarios requires explicitly modeling the output of the system. None of the existing approaches supports explicit specification of output functions.

The variety of modeling tasks that can be tackled to a large extend depends on the parameter estimation technique that is used. LAGRAMGE 2.0, IPM, and HIPM use the local optimization method ALG-717 (Bunch et al., 1993) to solve the parameter estimation task for each candidate model structure. Local optimization algorithms are unsuitable for estimating the parameters of nonlinear models with many parameters. The parameter space of such a model forms a large and multimodal landscape and finding the global optimum presents too hard of a problem for a local algorithm. Global optimization methods (Horst et al., 2000) and in particular metaheuristic methods are more robust in finding solutions in large and complex parameter spaces.

1.3 Contributions

The main contributions of the dissertation are summarized as follows:

1. **Development of a formalism for representing process-based models and background knowledge.** The process-based formalism is designed to alleviate the shortcomings of previous approaches. The main improvements integrated into the formalism are the following:

   (a) **A mathematical notation for all concepts and relations in the formalism.** Each concept for which there is a textual representation in the process-based formalism, including the constructs both in the models and the background knowledge, has its mathematical counterpart. The information contained in the models and the background knowledge is represented as a set of mathematical functions and relations. This notation is the means for formally expressing the properties of process-based models and background knowledge, as well as the algorithms that operate on them.

   (b) **Support for compartmental modeling.** The process-based formalism includes *compartments* as first-class citizens enabling the construction of hierarchical multi-compartment models.

   (c) **Incompletely specified process-based models.** Incompletely specified process-based models are process-based models which are incomplete in the structural specification or in the numerical parameters. Incomplete models serve as a means to specify the partial knowledge about the system being modeled, thus defining the task and space of feasible model candidates for the IPM methodology.

2. **Development of the IPM platform ProBMoT.** ProBMoT implements the proposed process-based formalism as the language for representing models, incomplete models, and libraries of background knowledge. Furthermore, ProBMoT implements the complete pipeline of inductive process modeling including the enumeration of all candidate model structures and parameter estimation. The main improvements implemented in ProBMoT are the following:

   (a) **Support for arbitrary observational scenarios.** ProBMoT includes an explicit output specification which includes not only observable and hidden variables, but arbitrary algebraic equations (including parameters) that can involve model variables and constants.

   (b) **Improved performance of the parameter estimation stage**. In addition to local optimization methods, ProBMoT integrates global optimization methods, based on metaheuristic algorithms, for the parameter estimation stage. These global optimization methods outperform local optimization methods in finding parameter values.

   (c) **Selection of different quality criteria.** ProBMoT integrates different quality criteria in the form of various error function. These error functions can serve as objective functions for the optimization algorithms for parameter estimation.

3. **Practical evaluation of the ProBMoT platform.** ProBMoT was experimentally evaluated on a case study of modeling phytoplankton growth in four different aquatic ecosystems. The case study aims to :

   (a) **Show the suitability of ProBMoT platform for automated modeling of dynamical systems.** Mathematical modeling of aquatic ecosystems comprises a considerable amount of knowledge reflected through a vast variety of different models that can be found in literature.

   (b) **Introduce criteria for comparing parameter estimation methods in the context of structure identification.** The criteria try to answer the following three questions. What is the difference between the errors of the best models obtained with different parameter estimation methods? What is the overall difference between the errors of the models obtained with different parameter estimation methods? What is the difference between the errors of each candidate model structure obtained with different parameter estimation methods?

   (c) **Demonstrate the superiority of global optimization methods over local optimization methods.** Numerous studies have confirmed the advantages of metaheuristics for parameter estimation when calibrating a single model. We extend those claims to the task of automated modeling and demonstrate that metaheuristics outperform local search when dealing with multiple candidate model structures.

1.4 Organization

This chapter provides a broader context for the thesis. It gives an overview of related areas and the research performed therein. It identifies the shortcomings of existing approaches and presents the motivation of the performed research. It concludes with a summary of the main contributions of the dissertation.

Chapter 2 briefly introduces inductive process modeling as an approach to automated modeling of dynamical systems. It discusses the most closely related approaches to the one proposed in the thesis. The discussion is structured along several design issues concerning representation and induction of process-based models.

Chapter 3 introduces and describes in detail the representational aspects of our approach to process-based modeling. It demonstrates the qualitative and quantitative aspects of process-based models. It describes the main components of process-based models: entities, processes, and compartments. Process-based models and background knowledge are introduced through a series of examples, each one building upon the previous ones.

Chapter 4 formalizes the process-based modeling constructs introduced in Chapter 3. Process-based models and libraries of background knowledge must satisfy a number of rules and constraints in order to form valid representations. This chapter introduces a formal notation suitable for precise, mathematically sound, definitions of the key concepts of the process-based formalism. The use of this formal notation and the provision of formal definitions simplify the task of expressing the properties of process-based models and libraries.

Chapter 5 starts with a discussion of how the qualitative aspects detailed in the previous chapter are translated into quantitative characteristics of process-based models. It then focuses on structural and parameter incompleteness of process-based models and how these are represented in the formalism. Next, it presents in detail incompletely specified models and discusses their use as constraints on the resulting model candidates.

Chapter 6 presents ProBMoT (**Pro**cess-**B**ased **Mo**deling **T**ool). ProBMoT uses the process-based formalism presented in Chapters 3 and 4 for describing the domain knowledge organized in libraries. The chapter starts with a definition of the inductive process modeling task. It then outlines the ProBMoT pipeline consisting of two main phases: model structure enumeration and parameter estimation. It then discusses each of these phases in detail and presents the implemented algorithms.

Chapter 7 presents the empirical evaluation of ProBMoT on the case study of modeling phytoplankton dynamics in four aquatic ecosystems. The aim of this study is to compare the influence of two established methods for parameter estimation, a local (ALG-717) and a global (DASA), on the overall process of automated modeling of aquatic ecosystems. First, the developed library for modeling aquatic ecosystems, which is used as domain knowledge, is presented. Next, the data from four ecosystems, relevant for modeling phytoplankton dynamics, and the conceptual models for each ecosystem are discussed. The results of the study are then presented and analyzed according to three evaluation criteria.

Finally, Chapter 8 concludes the dissertation. It summarizes the presented work, focusing on its contributions to science. It also outlines several directions for further work.

2 Inductive Process Modeling

Inductive process modeling (IPM) involves constructing a process-based model of a dynamical system from observed data about the system (capturing the behavior of the system over time). Process-based domain knowledge is also used in IPM, in addition to the observed data. The proponents of inductive process modeling claim that casting the domain and learned knowledge in terms of processes with associated equations is desirable for scientific and engineering domains, where such notations are commonly used.

Models in science and engineering often provide explanations which include variables, objects, or mechanisms that are unobserved, but help predict the behavior of the observed variables. Moreover, explanations posit causal structures that link these elements to observed variables. They often use general concepts or relations that occur in different models.

The term inductive process modeling was coined by Langley et al. (2002), who formally stated IPM as a challenging research problem for machine learning. The origins of the IPM idea, however, come from Džeroski and Todorovski (2001, 2002), who considered the use of the paradigm (without naming it IPM) for modeling population dynamics processes (Džeroski and Todorovski, 2003). The first working machine learning system for IPM was thus LAGRAMGE 2.0 (Todorovski, 2003; Todorovski and Džeroski, 2006).

Bridewell et al. (2008) give a more detailed account of the paradigm of inductive process modeling. They describe a process-based formalism for representing models and domain knowledge. They also describe a system for learning process-based models and its experimental evaluation. An extension of IPM, called HIPM, was introduced by Todorovski et al. (2005), which organizes the process-based models and domain knowledge in a hierarchical manner.

In the remainder of this chapter, we first briefly introduce inductive process modeling, following Bridewell et al. (2008). We then give a brief overview of the systems LAGRAMGE 2.0, IPM and HIPM, discussing the design decisions taken in each of them concerning the representation and induction of process-based models. We identify their limitations and shortcomings that have motivated the work presented in this dissertation.

2.1 Process-Models and the IPM Task

Bridewell et al. (2008) define a process model as a set of processes that link observable variables with each other causally, possibly through unobserved theoretical terms. The processes are represented in terms of differential equations (for modeling change over time) and algebraic equations (for modeling instantaneous effects). The specific processes in such a model are assumed to be instances of some set of generic processes that, taken together, constitute the background knowledge about the domain in question. Generic processes can be viewed as general laws and the model, which incorporates some of the situational conditions, can be viewed as the formal explanation of the observed behaviors (trajectories) of the system.

Process models put an emphasis on comprehensibility and plausibility by using a modeling notation familiar to scientists. They do introduce some (minimal) syntactic overhead, as one could build a more concise model by putting all equations into a single process. Having, however, the equation fragments assigned to specific processes, clarifies the mechanisms of the modeled system.

Bridewell et al. (2008) identify that the direct mapping between the model's equations and the background knowledge provides two benefits. First, the explicit relationship between the two levels of knowledge removes the need to infer such a mapping, which can be difficult even for domain scientists. Second, the knowledge encoded in the generic processes provides an important source of constraints on the specific models one might consider as explanations for a data set.

Inductive process modeling involves constructing a process-based model of a dynamical system from observed behavior(s) of the system. Process-based domain knowledge is also used, in addition to the observed data. The task of inductive process modeling is thus defined as:
Given:

- Observations for a set of continuous variables as they change over time,
- A set of observed, unobserved, and exogenous (forcing) variables that the model includes,
- Generic processes that specify causal relations among variables using generalized functional forms,
- Constrains, such as variable type information, that determine which processes may relate particular variables.

Find:

- A specific process model that, when given initial values for the modeled variables and values for any exogenous variables, explains the observed data and predicts unseen data accurately.

2.2 Design Decisions in Inductive Process Modeling

A platform for solving the inductive process modeling task must make several design decisions. The first major aspect that any platform has to address is the representation of the models and background knowledge, as well as the representation of the specification of the modeling task. The second major aspect is the methodology for induction of models, in which the two main components are the searching over the space of candidate model structures and the estimation of model parameters.

2.2.1 Representation

The representation of models and background knowledge can be implemented in a number of ways. The simplest way is to implement the components of the model as objects in an object-oriented language. This would relieve the developer of such a platform from devising an interface for communicating models and knowledge into and out of the platform. It would, however, put the burden on the user of the platform, who would have to be familiar with the programming language of choice.

Another option is to represent the models and background knowledge in a machine readable textual format, such as XML. XML representations can be easily read by humans, but are still cumbersome for use by domain experts.

The most acceptable and intuitive representation for domain experts would be a textual representation with simple syntactic rules and minimal artificial constructs. Such a representation would require the development of a domain specific language (DSL) for representing models and background knowledge and an accompanying parsing infrastructure.

Models

The representation revolves around the notion of a process as an abstraction for the phenomena that occur in the system. A process carries a quantitative description in the form of a set of equations that are associated with it. In addition to processes, the model contains variables and constant parameters. The variables can be considered as first-class citizens of the model, or can be grouped to form larger constructs, such as entities, which would relate to actors of the system. Some processes may be viewed as compound processes, which can be decomposed into simpler nested processes, thus forming a containment hierarchy of processes. A representation format may also allow for decomposition of models into sub-models, to provide means for managing complex models.

Background Knowledge

Domain-specific knowledge is of different types, including background knowledge and constraints (Muggleton, 1991, 1999; Džeroski et al., 2010). The background knowledge for inductive process modeling is structured into a domain-specific library, that contains definitions of entity and process templates. The library contains alternative mathematical formulations for the conceptual processes. The more background knowledge (more alternative formulations), the larger the potential space of models to consider.

The representation choices for the library should allow the domain experts to express as much of their knowledge as possible. Expressing knowledge about individual processes is crucial, but it is important to be able to express relations between variables/entities and processes, as well as between processes.

Task Specifications

Along with the background knowledge, the user of an inductive process modeling platform must provide a task specification. The task specification places constraints on the search space of models: The more constraints, the smaller the actual search space.

The task specification is formulated as a configuration of the particular system of interest. The minimal information present in the task specification concerns the variables/entities that are present in the system. If no other information is passed to the platform, then all viable alternatives encoded by the library can be considered as plausible models. Oftentimes, the domain expert has additional knowledge about the system at hand, in the form of processes that previous research has shown to occur, or previous analysis of the numerical parameters, and is not interested in looking for alternatives for them.

2.2.2 Induction

The induction algorithm is at the core of inductive process modeling. The background knowledge and task specification define the space of candidate model structures. The induction algorithm searches this space and estimates the constant parameters of each candidate model structure.

Search Over the Space of Model Structures

The space of model structures is determined by the library of background knowledge and the task specification. The domain-specific library acts as a data base of alternative modeling components. The more components it contains, the larger the search space is. It is important, however, that this space restricts implausible model structures as much as possible, thus focusing the search resources on the important candidate models.

The task specification restricts the space of candidate model structures by providing constraints about the modeled system. The more detailed the task specification, the more constraints on the candidate models are provided. This implies a smaller space of candidate models.

The search space can be explored exhaustively or non-exhaustively. Exhaustive enumeration is feasible when the size of the search space is small, and allows us to make stronger statements about the results (given that we have not missed any candidate model). Non-exhaustive search is usually heuristically driven, and allows us to explore larger search spaces, but may miss some good candidate models.

Parameter Estimation

Each candidate model that is considered during induction is fitted against measured data. The parameter estimation stage determines how well each candidate model will fit the data, and in the end, how well the best model will fit the data. Inductive process modeling platforms evaluate a large number of candidate models during the search, which in turn implies the need for a general-purpose, robust parameter estimation technique that can cope with many different error landscapes that may be encountered during the search.

2.3 LAGRAMGE 2.0

LAGRAMGE 2.0 (Todorovski, 2003; Todorovski and Džeroski, 2006) is a modeling framework that integrates the knowledge-based (theoretical) approach to modeling with data-driven (empirical) modeling. The framework allows for integration of modeling knowledge specific to the domain of interest in the process of model induction from measured data. The knowledge is organized around the central notion of basic process in the domain and it includes models thereof as well as guidelines for combining models of individual processes into a model of the entire observed system. This framework uses the equation discovery tool LAGRAMGE (Todorovski and Džeroski, 1997) to heuristically search through the space of candidate models, match them against data, and find the one that fits the data best.

LAGRAMGE 2.0 uses a special-purpose textual representation for models and background knowledge with minimal syntactic overhead. The background knowledge consists of a taxonomy of variable types, taxonomy of process classes and a set of combining schemes. The taxonomy of variable types includes a declaration of all valid variable types and their super-types. The taxonomy of process classes is defined in such a way that it specifies that the process model can be used for modeling processes in the current class as well as processes from the more general (ancestor) classes in the taxonomy.

Each definition of a process class consists of: the type of variables involved, conditions on the variables involved, and a declaration of the process model. The first part of the definition specifies the types of variables that can influence and be influenced by processes in the class. The second part of the process class definition specifies constraints on the variables involved in the process, usually denoting that the different arguments of a process cannot contain the same variable. The final part of the process class definition specifies the equation template that is used by domain experts to model processes in the class.

The library also includes one combining scheme for each type of variables which is being modeled with LAGRAMGE 2.0. The combining scheme specifies how to build the equation that models the time change of a system variable from individual process models. The combining scheme usually uses aggregation functions, such as sum or product, to combine the models of all processes that influence that variable.

In order to use the knowledge for modeling of a particular system, a specification of the system is provided. The specification includes a list of system variables and their associated types. It also includes a list of processes, and their classes, that govern the dynamics of the observed system.

The domain-specific knowledge is transformed into a context-dependent grammar based on the declarations of the system variables and processes. The starting symbol for the grammar corresponds to the combining schemes. The other nonterminal symbols in the grammar correspond to the processes classes. Alternative productions for each nonterminal symbol specify alternative models of the corresponding process class.

The resulting grammar specifies the space of candidate models for the observed system. The equation discovery method LAGRAMGE is then used to search through the space of candidate models. LAGRAMGE can use either an exhaustive search, limited by the depth of the parse trees, or a beam search with adjustable beam width.

LAGRAMGE 2.0 substitutes the downhill simplex and Levenberq-Marquart algorithms that LAGRAMGE uses with ALG-717, a non-linear optimization algorithm, proposed by Bunch et al. (1993). ALG-717 can take into account lower and upper bounds on parameter values. In addition, LAGRAMGE 2.0 restarts the parameter estimation method with different randomly chosen combinations of initial values, thus decreasing the likelihood of getting stuck in a local optimum.

LAGRAMGE 2.0, however, suffers from a number of shortcomings. It does not allow the user to fix the values of constant parameters in modeling scenarios where their value is known. The only workaround is to fix the values of the constant parameters in the library, thus defeating the purpose of the library as a repository of domain-wide knowledge. Moreover, a constant is not defined at any single place, but its definition is taken to be at the place where it is used. That makes fixing the value of a constant even more tedious, since one constant can have multiple occurrences in the library, and they all have to be updated. Difficulties in understanding and interpretation of libraries and models can arise from unnamed constants which are used in the combining schemes. These are all given a generic name, which makes it hard to track their meaning in large models, where they appear at multiple locations.

The background knowledge about processes and their influence on variables is divided across the taxonomy of process classes and the combining scheme. The need for using a combining scheme arises primarily because each process can return only a single equation fragment. The combining scheme is then used to attribute this equation fragment to multiple variables that might be targeted by the process. Allowing processes to influence multiple variables will overcome the need for using a cumbersome combining scheme.

Finally, even though LAGRAMGE 2.0 improves upon LAGRAMGE's parameter estimation, it still relies on a local search method. The parameter space is often highly non-linear with multiple local optima. While the multiple restarts technique alleviates this problem as compared to a single run of a local optimization method, it still does not address it properly.

2.4 IPM

The background knowledge in IPM (Bridewell et al., 2008) is specified in terms of generic processes. They do not commit to particular variables or parameter values, but they indicate constraints on them, like the type of the variables and the range of the parameters. The background knowledge also contains a hierarchy of variables that descend from the same base type *number*.

In addition to the generic processes, IPM needs a set of typed variables and training data for the exogenous and observable system variables. Combined with the set of generic processes, this information defines the space of model structures that the IPM will search.

To produce a simulation of a process model, IPM carries out two phases. The first phase combines the component from each of the processes into a system of differential and algebraic equations. During this conversion, the system ensures that the algebraic equations will be solved according to their causal ordering. The module's second phase evaluates the ODE model using CVODE (Hindmarsh et al., 2005), a solver for first-order differential equations, coupled with basic arithmetics for handling the algebraic equations.

IPM operates in three stages. In the first stage, it finds all permissible instantiations of the generic processes with the specified variables. A permissible instantiation takes into account only the variables whose types are compatible with the generic process. In the second stage, subsets of this collection of partially instantiated processes are used to form generic models, each of which specifies an explanatory structure. The space of generic models is a power set of the set of partially instantiated processes. The system carries out an exhaustive search of model structures by enumerating this power set, retaining as candidate models only those members that satisfy user-provided constraints, including the maximum number of processes in the model and the list of generic processes that must be instantiated. The third stage infers the parameter values for each generic model using ALG-717 (Bunch et al., 1993) which carries out a second-order gradient descent search through the parameter space.

A major limitation of the knowledge representation in IPM is the inability to exclude combinations of processes when considering candidate model structures. Because the space of candidate models is generated as a power set, any combination of generic processes is considered plausible. However, the library typically contains many alternatives of the same conceptual process and IPM, therefore, takes into consideration models which contain duplicate conceptual processes. This does not correspond to the expectations of domain experts, who would consider as plausible only models that contain one instance of each conceptual process.

Another limitation is the inability to imply that certain processes may only appear in conjunction with some other particular process or not appear at all. This issue arises in IPM's treatment of nested processes. IPM allows the modeler to decompose a complex process into several simpler and more manageable processes that communicate information through some variables. These component processes, however, make sense only when they appear in the model together with the other nested processes that make up the complex interaction. The model that results when some of the nested processes are missing is not considered as plausible by the domain expert.

In a similar fashion to LAGRAMGE 2.0, IPM also relies on ALG-717 for parameter estimation, which gives rise to the same concerns over the ability of the method to find globally acceptable parameter values.

2.5 HIPM

HIPM (Todorovski et al., 2005) extends IPM's formalism and organizes process knowledge in a hierarchical manner. Hierarchical processes characterize an observed system's behavior at distinct levels. The hierarchical structure lets the induction method carry out search through an AND/OR space rather than an OR space, thus reducing the number of candidate models considered and ensuring that these models will make sense to domain scientists. This approach requires greater effort to encode background knowledge, but offsets this with a more efficient search over a space of more plausible models.

In HIPM's formalism, variables are replaced with entities that group properties of the observed actors of the system. Entities can have two kinds of properties—variables and constants. The value of variables can change over time, while constant parameters describe aspects of an entity that do not change over time for a given system.

HIPM identifies and addresses two limitations of IPM, regarding the assumptions how to combine processes into a model. The first invalid assumption that IPM makes is that it can combine any set of generic processes to produce a valid model structure. This assumption leads to an underconstrained model space containing many candidates that violate the domain expert's expectations. The second assumption regards all process influences as additive, which is unrealistic in many scenarios.

The interleaved structure-type hierarchy of generic processes places two types of constraints on the space of candidate models. The hierarchy of process types defines mutually exclusive modeling alternatives, whereas the subprocess hierarchy defines correct model structures in terms of the minimal set of necessary component sub-models. This organization contrasts IPM's representation of models as a flat collection of generic processes, which can be combined arbitrarily into candidate model structures.

HIPM uses heuristic beam search and knowledge-guided refinement operators to navigate the model space. The system takes as input a hierarchy of generic processes, a set of entities with associated variables, a set of observed trajectories of the variables and a specification of the beam width. On each beam-search iteration, HIPM refines the current selection of models by one step, adding the non-redundant structures back to the beam. In the first iteration, the system generates all permitted models that exclude any optional processes. In subsequent iterations, the refinement operator would add a single optional process to the current model structure, which may require the addition of multiple processes, depending on the background knowledge.

HIPM, like its predecessors, does not provide support for multi-compartmental modeling, or other means for (spatial) structuring of complex models. In a multi-compartment model, the model is hierarchically segregated in compartments. Each compartment denotes one (spatial) unit of the system. Many of the compartments can be of similar nature thus containing the same entity and process types. In order to implement a multi-compartmental scenario in HIPM (as well as in LAGRAMGE 2.0 and IPM), one has to place all entities and processes at the model level and distinguish the pertaining compartments by carefully naming the entities and processes.

HIPM does not introduce a modeling language for the representation of models and background knowledge. Instead, the models and background knowledge are expressed in its internal representation, directly in Python. While Python is a language with very little syntactic overhead and manages to capture the semantic relations of process-based modeling in a fairly clean manner, it is nevertheless a programming language. Having in mind that inductive process modeling tools are intended to be used not only by computer scientists, but even more so by domain experts from diverse fields (including environmental sciences, computational biology, and earth sciences among others), a conventional programming language is not the optimal solution.

Similarly to LAGRAMGE 2.0 and IPM, HIPM uses ALG-717 with random restarts for parameter estimation and suffers from its limitations.

3 Representing Process-Based Models and Background Knowledge

Process-based modeling takes a knowledge-based approach to modeling. Process-based models (Bridewell et al., 2008) use a two-level representation combining qualitative and quantitative information. At the qualitative level, a process-based model consists of entities, which correspond to the main actors of the modeled system, and processes, which correspond to relations between entities. At the quantitative level, each entity is described in terms of variables and constants that represent its properties, and each process is represented as a set of equations, algebraic or differential, that quantify the relations between the entities. The equations from all the processes in the model can be compiled to obtain a system of ordinary differential equations, which is the ultimate quantitative representation of the system.

The two levels of knowledge representation used for process-based modeling allow for specifying the models at different levels of abstraction. At the qualitative level, an abstract view of the modeled system is represented, depicting only the key components of the system and the relations between them. At the quantitative level, a detailed view of the system is represented, which is equivalent to a system of ordinary differential equations that can be used for further quantitative analysis of the system.

Process-based models integrate the explanatory aspect of qualitative models with the quantitative aspect of differential equations that allow for effective simulation and prediction of the behavior of the system. When dealing with dynamical systems, scientists and engineers often refer to processes that govern system dynamics and entities that are influenced by those processes. Processes causally link system variables and entities, possibly through unobserved theoretical terms. The qualitative abstraction level of model representation corresponds to the explanatory aspect of the process-based models. On the other hand, to allow for a quantitative analysis of system behavior, process-based models specify a quantitative model for each process; when put together, these models yield a complete model of the system that takes the form of a system of ordinary differential equations.

3.1 Process-Based Models and Their Components

Process-based models consist of two basic types of elements: entities and processes. Entities represent the actors of the observed system. These actors are involved in processes that explain how entities interact, as well as what is the influence of the interactions on the involved entities themselves. When we deal with equation-based models, entities correspond to the variables in the equations and processes to arithmetical expressions (equation fragments).

The state of the modeled system is represented as a set of entities. Each entity corresponds to one object that appears in the system. If we consider a simple lake ecosystem, nutrients (such as *phosphorus* and *nitrogen*), as well as *phytoplankton* would be represented as entities. The phenomena that occur in the system would be described by the processes of the model. Each process in the model corresponds to a single phenomenon in the system. For instance, the *growth* of *phytoplankton* (limited by *nitrogen* and *phosphorus*) would be a process. This simple relation is represented in Figure 3.1(a).

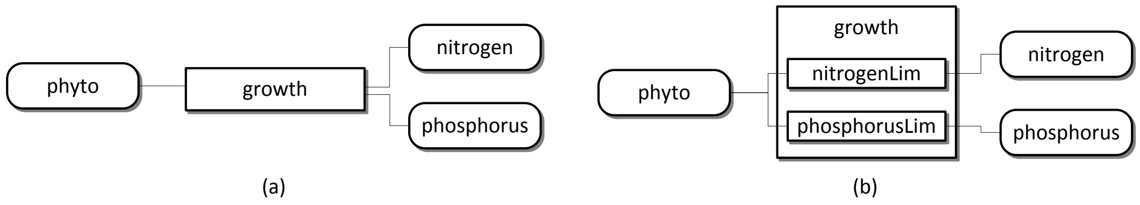


Figure 3.1: An example model of a lake ecosystem, described with (a) only one top-level process or (b) with nested processes.

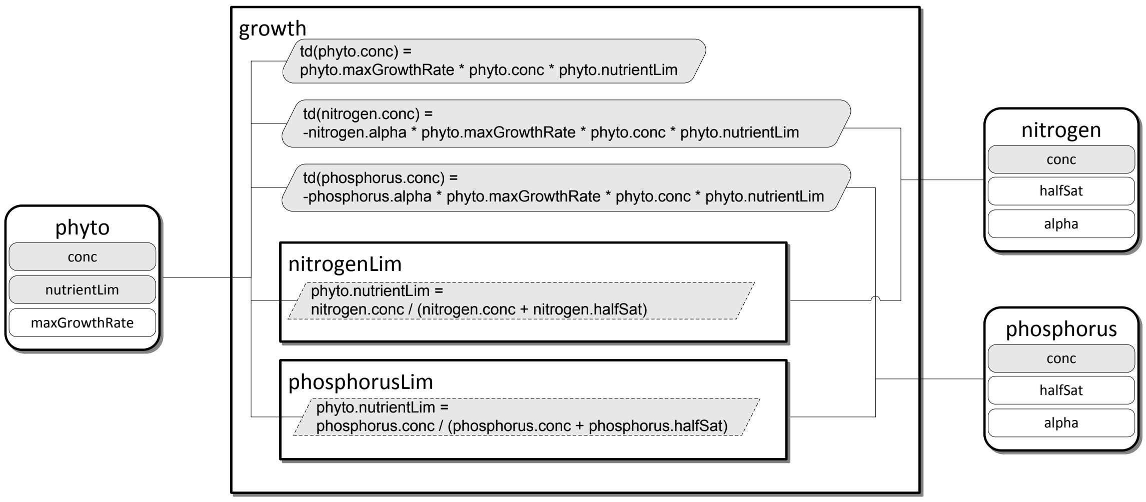

growth
td(phyto.conc) =
phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
td(nitrogen.conc) =
-nitrogen.alpha * phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
td(phosphorus.conc) =
-phosphorus.alpha * phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
nitrogenLim
phyto.nutrientLim =
nitrogen.conc / (nitrogen.conc + nitrogen.halfSat)
phosphorusLim
phyto.nutrientLim =
phosphorus.conc / (phosphorus.conc + phosphorus.halfSat)
phyto
conc
nutrientLim
maxGrowthRate
nitrogen
conc
halfSat
alpha
phosphorus
conc
halfSat
alpha

Figure 3.2: A quantitative representation of the example model of a trophic relation in a lake ecosystem.

Some processes that occur in the system can be more easily described in terms of several simpler processes. We consider the later processes to be nested processes, because they are nested within more complex enclosing processes. In our simple lake ecosystem, the phytoplankton *growth* process can be defined in terms of two nested processes that capture the limitation of phytoplankton growth by nitrogen and phosphorus, respectively. These relations between the entities and processes in the system are depicted in Figure 3.1(b).

In addition to the purely qualitative relations (such as those depicted in Figure 3.1), process-based models provide quantitative information about the phenomena they represent. Each entity is described with a number of properties. Some of them do not change their value through time and are referred to as constants. Others, which change their value with time, are called variables. For example, the *phytoplankton* entity has two variables, *concentration* and *nutrient limitation*, and one constant property of *maximal growth rate*. Furthermore, the *nitrogen* entity can be specified with its *concentration* (variable) and the two constants of *half saturation* (corresponding to the concentration at which the process rate halves) and *alpha* (stoichiometric ratio between algal biomass and the nutrient). The same variables and constants are used to describe the *phosphorus* entity.

A process definition provides quantitative description of the relation it represents in the form of one or more equations. Each equation can contain only properties (variables and constants) of those entities that are involved in the process of interest. For example, the *nitrogen limitation* process, which involves the *nitrogen* and *phytoplankton* entities, can only contain references to the variables and constants of *nitrogen* or *phytoplankton*. These quantitative relations are presented in Figure 3.2.

Example 3.1 shows the representation of the entities of the simple lake ecosystem conceptualized in Figure 3.2 in the formalism of process-based models. The example contains

Example 3.1: Formal representation of the entities in Figure 3.2.

```
entity phyto {
    vars:
        conc{role: endogenous; aggregation: sum; initial: 10},
        nutrientLim{aggregation: product};
    consts:
        maxGrowthRate = 0.5,
}
entity phosphorus {
    vars:
        conc{role: endogenous; initial: 3};
    consts:
        halfSaturation = 0.02,
        alpha = 0.1;
}
entity nitrogen {
    vars:
        conc{role: exogenous};
    consts:
        halfSaturation = 0.2,
        alpha = 0.7;
}
```

Example 3.2: Formal representation of the top-level process shown in Figure 3.2(a).

```
process growth(phyto, phosphorus, nitrogen) {
    equations:
        td(phyto.conc) = phyto.maxGrowthRate ∗ phyto.conc ∗ (phosphorus.conc / (
            phosphorus.conc + phosphorus.halfSaturation) + nitrogen.conc / (nitrogen.conc
             + nitrogen.halfSaturation)),
        td(phosphorus.conc) = phosphorus.alpha ∗ phyto.maxGrowthRate ∗ phyto.conc ∗
            phosphorus.conc / (phosphorus.conc + phosphorus.halfSaturation);
}
```

three entities: *nitrogen*, *phosphorus* and *phytoplankton*(*phyto*). Each entity is specified with a list of its variables (*vars*) and and a list of its constants (*consts*). These are the properties of the entity that are important in the given modeling context.

For each variable, we specify an aggregation function, i.e., the method of aggregating the influences of different processes on that variable. The *nutrientLim* variable of the *phytoplankton* entity, for example, has a *product* aggregation function specifying that the multiple influences on the *nutrientLim* variable are to be multiplied.

Furthermore, each variable is assigned a *role*, which can be either *exogenous* or *endogenous*. Exogenous variables are used as forcing variables and as such are considered external to the system and not modeled within it. Endogenous variables, on the other hand, are variables which are modeled within the system. They describe the state of the system and appear in the system of ODEs. In addition, endogenous variables can have *initial* values, which designate the value of the variable at the initial time point.

Constants are defined in a straightforward manner, by assigning a numerical value to them.

Example 3.2 presents the main process of the system, the growth of phytoplankton, Example 3.3: Formalized representation of the processes in Figure 3.2 organized as nested processes.

```
process growth(phyto, [phosphorus, nitrogen]) {
    processes:
        phosphorusLim, nitrogenLim;
    equations :
        td(phyto.conc) = phyto.maxGrowthRate ∗ phyto.conc ∗ phyto.nutrientLim,
        td(phosphorus.conc) = phosphorus.alpha ∗ phyto.maxGrowthRate ∗ phyto.conc ∗
            phyto.nutrientLim;
}
process phosphorusLim(phyto, phosphorus) {
    equations:
        phyto.nutrientLim = phosphorus.conc / (phosphorus.conc + phosphorus.
            halfSaturation);
}
process nitrogenLim(phyto, nitrogen) {
    equations:
        phyto.nutrientLim = nitrogen.conc / (nitrogen.conc + nitrogen.halfSaturation);
}
```

limited by phosphorus and nitrogen, corresponding to Figure 3.1 (a). The growth process influences the phytoplankton concentration positively and the phosphorus concentration negatively since phosphorus is consumed in the process. Nitrogen is also consumed, but because its concentration is an exogenous variable, it is not modeled with an equation within the system of ODEs. The two variables which are affected by this process are influenced by differential equations. Note that the symbol *td(x)* denotes a *time derivative* of the variable *x*.

Example 3.3 shows the growth of phytoplankton modeled using two nested processes as the conceptual model from Figure 3.1 (b). Instead of being separate arguments to the growth process, phosphorus and nitrogen form an argument set (indicated by the square brackets). The influences of phosphorus and nitrogen as limiting factors of the phytoplankton growth are expressed with the two processes of *phosphorusLim* and *nitrogenLim*, which are nested in the growth process. Both nutrient limitations are modeled using a Monod (saturation) function and captured within the *nutrientLim* variable. Expressing relations with nested processes improves the modularity of the model, making it easier to add or remove arguments of the process because their influence is well encapsulated and clearly expressed.

The process-based model can be 'flat' or 'structured'. If the model is flat, all of the entities and processes are at the top level of the system (which is the only level). The system can also be structured through the use of compartments, organized in a tree-shaped hierarchy. Compartments are containers for structuring a system. A system can contain one or more compartments, or none at all, and each compartment can contain other compartments as sub-compartments. A compartment contains entities and processes, which is its primary function. Hence, compartments form a tree-shaped hierarchy with the whole system at the top, which itself can be considered as a top-level compartment.

Example 3.4: A compartmental process-based model of a lake ecosystem.

```
compartment epi {
    entity phyto {
        vars:
            conc{role: endogenous; aggregation: sum; initial: 10},
            nutrientLim{aggregation: product};
        consts:
            maxGrowthRate = 0.5,
    }
    entity phosphorus {
        vars:
            conc{role: exogenous};
        consts:
            halfSaturation = 0.02,
            alpha = 0.1;
    }
    process growth(phyto, [phosphorus]) {
        processes:
            phosphorusLim;
        equations :
            td(phyto.conc) = phyto.maxGrowthRate ∗ phyto.conc ∗ phyto.nutrientLim,
    }
    process phosphorusLim(phyto, phosphorus) {
        equations:
            phyto.nutrientLim = phosphorus.conc / (phosphorus.conc + phosphorus.
                 halfSaturation);
    }
}
compartment hypo {
    entity phyto {
        vars:
            conc{role: endogenous; aggregation: sum; initial: 10},
            nutrientLim{aggregation: product};
        consts:
            maxGrowthRate = 0.5,
    }
    process mortality(phyto) {
        consts:
            mortRate = 0.1;
        equations:
            td(phyto.conc) = mortRate ∗ phyto.conc;
    }
}
entity environment {
    vars:
        flow{role: exogenous};
    consts:
        epiVolume = 6000000,
        hypoVolume = 20000000;
}
process mixing(epi.phyto, hypo.phyto, environment) {
    equations:
        td(epi.phyto.conc) = environment.flow / environment.epiVolume ∗ (epi.phyto.conc
             hypo.phyto.conc),
        td(hypo.phyto.conc) = environment.flow / environment.hypoVolume ∗ (hypo.phyto.
             conc  epi.phyto.conc);
}
```

Example 3.4 contains a compartmental model of the simple lake ecosystem. The model contains two compartments, *epilimnion* (*epi*) and *hypolimnion* (*hypo*), which represent layers in a thermally stratified lake. The upper layer—epilimnion—contains phytoplankton and phosphorus. The epilimnion is heated by solar radiation and contains enough light to support phytoplankton growth, during which phosphorus is consumed. Unlike in Example 3.3, here only phytoplankton concentration is modeled and phosphorus is taken as an exogenous variable. Therefore, the growth process contains only an equation for phytoplankton concentration. The lower layer—hypolimnion—contains only phytoplankton. This layer does not contain enough light to support phytoplankton growth and the only process which occurs in the hypolimnion is phytoplankton decay.

We include the properties of the environment in a separate entity at the top level of the model. This *environment* entity includes the *flow rate* between the epilimnion and hypolimnion, where positive values indicate flow from the hypolimnion to the epilimnion and negative values the other way around. The environment entity also contains the volumes of the two layers (*epiVolume* and *hypoVolume*) as constants.

An important aspect in compartmental modeling is that of inter-compartmental relations. Compartments serve the purpose of grouping and organizing entities and processes. Processes in each compartment are restricted to affecting the entities within the same compartment, including entities which are contained in a nested sub-compartment. The processes that represent interactions between entities from different compartments are placed at the same level as the interacting compartments. In Example 3.4, the *mixing* process represents an inter-compartmental interaction. It models the mixing of phytoplankton from the hypolimnion and the epilimnion as a consequence of the natural flow of water from one layer to the other.

For each endogenous variable x in the model entities, we compile one equation with that variable on its left-hand side. The equation is compiled by combining the effects of all equations in the model that influence x, i.e., all equations which have x on the left-hand side. The aggregation function used for combining the equations is specified in the definition of x. In Example 3.3, the variable *phyto.nutrientLim* (the variable *nutrientLim* of the entity *phyto*) is influenced by two equations, those from the processes of *nitrogenLim* and *phosphorusLim*. Having in mind that the aggregation of the influences is performed by multiplication (see Example 3.1), we obtain the following equation for *phyto.nutrientLim*:

$$\begin{aligned} phyto.nutrientLim = {} & \frac{phosphorus.conc}{phosphorus.conc + phosphorus.halfSaturation} \\ & \times \frac{nitrogen.conc}{nitrogen.conc + nitrogen.halfSaturation} \end{aligned} \tag{3.1}$$

By compiling the equations for all endogenous variables, we get a system of ordinary differential equations (ODEs). This system of ODEs is a quantitative model of the dynamical system at hand. This model can then be used to perform model simulation and parameter estimation.

3.2 From Process-Based Models to Background Knowledge

Entities in a process-based model often share common properties. If we compare the *phosphorus* and *nitrogen* entities from Example 3.1, we can see that they share similarities with respect to their variables and constants. This is to be expected, since they are both nutrients. Properties which hold for a number of entities are specified through objects which we call *entity templates*. They are used for specifying common properties of entities. The template captures some general knowledge that holds for many different cases and can be reused when dealing with different specific scenarios. An entity template contains partial information for an entity that is general and can be used for generating many entity instances.

Similarly, if we compare the processes *nitrogenLim* and *phosphorusLim* from Example 3.3, we can see that they have equations that adhere to the same general pattern: they both represent processes of a saturated Monod-type nutrient limitation. Therefore, it makes sense to try to group such similar processes within some more general concepts. In analogy to entity templates, we introduce *process templates*—objects that represent parameterized recipes for creating processes. Process templates contain generic information that can be reused for generating a number of specific process instances.

Example 3.5: A library of background knowledge for modeling aquatic ecosystems.

```
library AquaticLibrary;
template entity Nutrient {
    vars:
        conc {aggregation: sum};
    consts:
        halfSaturation {range: <0,inf>},
        alpha {range: <0,inf>};
}
template entity PrimaryProducer {
    vars:
        conc {aggregation: sum},
        nutrientLim{aggregation: product};
    consts:
        maxGrowthRate {range: <0,inf>};
}
template process Growth(pp : PrimaryProducer, ns : Nutrient<1,inf>) {
    processes:
        NutrientLimitation(pp, <n:ns>);
    equations:
        td(pp.conc) = pp.maxGrowthRate ∗ pp.nutrientLim ∗ pp.conc,
        td(<n:ns>.conc) = n.alpha ∗ pp.maxGrowthRate ∗ pp.nutrientLim ∗ pp.conc;
}
template process NutrientLimitation(pp : Phytoplankton, n : Nutrient) {
    equations:
        pp.nutrientLim = n.conc / (n.conc + n.halfSaturation);
}
```

The set of entity and process templates relevant to a modeling domain of interest are collected into a *library* of background knowledge. Example 3.5 shows a background knowledge library for modeling aquatic ecosystems. This library generalizes the modeling notions used in the lake ecosystem model from Examples 3.1 and 3.3.

The library defines two entity templates, *Nutrient* and *PrimaryProducer*. They contain the appropriate definitions for the variables and constants present in the entities in Example 3.1. One important difference is that the constants in the library are not bound to particular numerical values, but instead specify a range of allowed values. Each instance of the entity template can take an arbitrary value from the specified range. Another important difference is that variables are not assigned roles in the library, because roles are assigned by a specific model. Depending on the particular system at hand, the modeling scenario, and the level of detail we want to model, a particular variable may be specified as endogenous, if modeling it is part of the task, or we cannot directly observe its values. Otherwise, if we are not interested in modeling the variable and we have sufficient measurements, it can be specified as exogenous.

In addition to the two entity templates, the simple lake ecosystem library defines two process templates: *Growth* and *NutrientLimitation*. The *Growth* process template gives

Example 3.6: A process-based model of a lake ecosystem based on the library of background knowledge from Example 3.5.

```
model lakeModel : AquaticLibrary;
entity phyto : PrimaryProducer {
    vars:
        conc {role: endogenous; initial: 10},
        nutrientLim;
    consts:
        maxGrowthRate = 0.5;
}
entity phosphorus : Nutrient {
    vars:
        conc {role: endogenous; initial: 3};
    consts:
        halfSaturation = 0.02,
        alpha = 0.1;
}
entity nitrogen : Nutrient {
    vars:
        conc;
    consts:
        halfSaturation = 0.2,
        alpha = 0.7;
}
process growthPhyto(phyto, [phosphorus, nitrogen]) : Growth {
    processes:
        phosphorusLim, nitrogenLim;
}
process phosphorusLim(phyto, phosphorus) : NutrientLimitation {}
process nitrogenLim(phyto, nitrogen) : NutrientLimitation {}
```

a recipe for defining processes of phytoplankton growth limited by a set of nutrients and has two arguments. The first argument is an entity of type *PrimaryProducer* named *pp*, which represents the phytoplankton whose growth is being modeled. The second argument is a set of entities of type *Nutrient* named *ns*, which represents the nutrients that limit the phytoplankton growth. The specification *<1, inf>* denotes that the set has to contain at lest one entity. The *Growth* process contains a nested process of type *NutrientLimitation* for each nutrient *n* from the set *ns* as specified by the declaration: *NutrientLimitation(pp, <n:ns>)*. It also contains one differential equation for the concentration of the phytoplankton, and one differential equation for each nutrient.

Having defined entity and process templates, we can use them to create suitable *entity* and *process instances*. Every instance acquires all of the properties which were specified in the template. Properties which are characteristic for the particular instance itself are specified within the instance definition. Using the library from Example 3.5, the simple lake model consisting of the entities from Example 3.1 and processes from Example 3.3 can be represented as shown in Example 3.6.

It is evident that by using the templates defined in the library we obtain a concise specification of the model. Instead of providing a complete specification of the entities and processes in the model itself, we create instances of the templates from the library. In the simple lake model, *phosphorus* and *nitrogen* are instances created using the *Nutrient* template, whereas *phyto* is created using the *PrimaryProducer* template. Each entity instance contains only the variables and constants which were specified in the respective template.

The *growth* process, on the other hand, is an instance of the process template *Growth*, having *phyto* as the first argument and the set of *nitrogen* and *phosphorus* as the second. The *growth* process has *phosphorusLim* and *nitrogenLim* as nested processes. Their definitions are very short, since all needed information is present in the *NutrientLimitation* process template. Each process instance can have as arguments entity instances of the types specified in the respective process template. In addition, a process instance acquires the equations from the process template, adjusted to use the variables and constants from the entity instances in the arguments.

In addition to entities and processes, compartments also follow the template-instance paradigm. A compartment template defines a type of compartment. It specifies the types of entities and processes which can be included in that type of compartment. Example 3.7 shows a library with compartments. It resembles the library in Example 3.5 as the definitions of *Nutrient*, *PrimaryProducer*, *Growth*, and *NutrientLimitation* are the same. It includes another entity template *Environment*, which captures some general properties of the lake. It also includes *Mortality* and *Mixing* processes.

The main difference between the libraries in Examples 3.5 and 3.7 is in the inclusion of two compartment templates *Epilimnion* and *Hypolimnion* in the latter. Epilimnion compartments can contain entities of type *Nutrient* and *PrimaryProducer* and processes of type *Growth* and *NutrientLimitation.* Hypolimnion compartments can contain only entities of type *PrimaryProducer* and processes of type *Mortality.* Processes of type *Mixing* are not contained in either type of compartments, because they represent inter-compartmental relations and have to be included at the model-level instead.

Example 3.7: A library for modeling aquatic ecosystems which includes compartments.

```
library AquaticCompartmentalLibrary;
template entity Nutrient {
    vars:
        conc {aggregation: sum};
    consts:
        halfSaturation {range: <0,inf>},
        alpha {range: <0,inf>};
}
template entity PrimaryProducer {
    vars:
        conc {aggregation: sum},
        nutrientLim{aggregation: product};
    consts:
        maxGrowthRate {range: <0,inf>};
}
template entity Environment {
    vars:
        flow;
    consts:
        epiVolume {range: <1,inf>},
        hypoVolume {range: <1,inf>};
}
template process Growth(pp : PrimaryProducer, ns : Nutrient<1,inf>) {
    processes:
        NutrientLimitation(pp, <n:ns>);
    equations:
        td(pp.conc) = pp.maxGrowthRate * pp.nutrientLim * pp.conc,
        td(<n:ns>.conc) = n.alpha * pp.maxGrowthRate * pp.nutrientLim * pp.conc;
}
template process NutrientLimitation(pp : Phytoplankton, n : Nutrient) {
    equations:
        pp.nutrientLim = n.conc / (n.conc * n.halfSaturation);
}
template process Mortality(pp : PrimaryProducer) {
        consts:
            mortRate {range: <1, inf>};
        equations:
            td(pp.conc) = mortRate * pp.conc;
}
template process Mixing(epiPP : PrimaryProducer, hypoPP : PrimaryProducer, env :
      Environment) {
    equations:
        td(epiPP.conc) = env.flow / env.epiVolume * (epiPP.conc  hypoPP.conc),
        td(hypoPP.conc) = env.flow / env.hypoVolume * (hypoPP.conc  epiPP.conc);
}
template compartment Epilimnion {
    entities:
        Nutrient, PrimaryProducer;
    processes:
        Growth, NutrientLimitation;
}
template compartment Hypolimnion {
    entities:
        PrimaryProducer;
    processes:
        Mortality;
}
```

We use compartment templates to create compartment instances. As compartments are containers for entities and processes, each compartment instance can only contain entity and process instances of the types specified in its compartment template. Example 3.8 shows the same model as in Example 3.4, reformulated to use the templates from the library in Example 3.7. As illustrated before, the model defined with the use of a library is much shorter and more concise.

3.3 Hierarchical Organization of Background Knowledge

Entity templates can be arranged into inheritance trees. The more general properties are placed in the entity templates which are higher up the tree. This enables the entity templates which are lower in the tree to inherit the properties of their ancestors and provides a modular, clean, and reusable design of the entity templates. Figure 3.3(a) shows how the entity templates from Example 3.5 can be arranged into an inheritance tree for the library. Note that, in Example 3.5, *Nutrient* and *Phytoplankton* shared the common variable concentration (*conc*), which has now moved to the *EcosystemEntity*. *Nutrient* and *Phytoplankton* are now subtypes of *EcosystemEntity* and thus inherit its properties, i.e., they inherit the *conc* variable.

In the model in Example 3.6, *nitrogenLim* and *phosphorusLim* have the same functional form—that of a Monod function. In practice, however, they can have any of a number of

Example 3.8: The compartmental process-based model of a lake (Example 3.4), expressed by using the template entities, processes, and compartments from a background knowledge library (Example 3.7).

```
model lakeCompartmentalModel : AquaticCompartmentalLibrary;
compartment epi : Epilimnion {
    entity phyto : PrimaryProducer {
        vars:
            conc{role: endogenous; initial: 10},
            nutrientLim;
        consts:
            maxGrowthRate = 0.5,
    }
    entity phosphorus : Nutrient {
        vars:
            conc{role: endogenous; initial: 3};
        consts:
            halfSaturation = 0.02,
            alpha = 0.1;
    }
    process growth(phyto, [phosphorus]) : Growth {
        processes:
            phosphorusLim;
    }
    process phosphorusLim(phyto, phosphorus) : NutrientLimitation {}
}
compartment hypo : Hypolimnion {
    entity phyto : PrimaryProducer {
        vars:
            conc{role: endogenous; initial: 10},
            nutrientLim;
        consts:
            maxGrowthRate = 0.5,
    }
    process mortality(phyto) : Mortality {
        consts:
            mortRate = 0.1;
    }
}
entity env : Environment{
    vars:
        flow{role: exogenous};
    consts:
        epiVolume = 6000000,
        hypoVolume = 20000000;
}
process mixing(epi.phyto, hypo.phyto, env) : Mixing {}
```

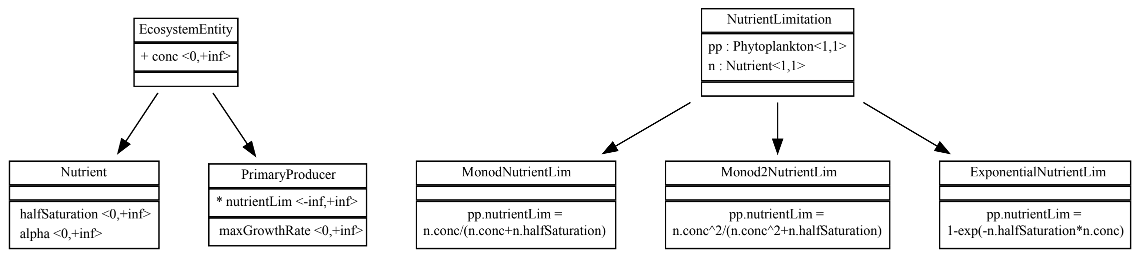


Figure 3.3: Hierarchies of templates in the aquatic ecosystem library. (a) A hierarchy of entity templates; (b) A hierarchy of process templates.

different functional forms, including Monod, Monod^2 or an exponential function. We can arrange the limitation processes into a hierarchy like the one presented in Figure 3.3(b) (Todorovski et al., 2005). The process template *NutrientLimitation* is at the root of the hierarchy. It does not supply any functional form by itself, but rather serves to organize the different nutrient limitation functions which are provided. Each alternative process template has *NutrientLimitation* as an ancestor in the inheritance tree of process templates.

Using this modeling approach, we can construct a library with an inheritance tree of entity types and process alternatives. Example 3.9 gives a hierarchical form of the aquatic ecosystem library. The corresponding process-based model is given in Example 3.10.

Example 3.9: A hierarchical version of the aquatic ecosystem library.

```
library AquaticHiearchicalLibrary;
template entity EcosystemEntity {
  vars : conc {aggregation: sum; unit: ”kg/mˆ3”; range: <0,inf>};
}
template entity PrimaryProducer: EcosystemEntity {
  vars:
    nutrientLim {aggregation: product};
  consts:
    maxGrowthRate{ range: <0,inf>; unit: ”1/(day)”;
}
template entity Nutrient : EcosystemEntity {
  consts:
    halfSaturation {range: <0,inf>},
    alpha {range: <0,inf>};
}
template entity Environment {
  vars:
    flow;
}
template process Growth(pp : PrimaryProducer, ns : Nutrient<1, inf>) {
  processes:
    NutrientLimitation(pp, <n:ns>);
  equations:
    td(pp.conc) = pp.maxGrowthRate ∗ pp.nutrientLim ∗ pp.conc,
    td(<n:ns>.conc) = n.alpha ∗ pp.maxGrowthRate ∗ pp.nutrientLim ∗ pp.conc;
}
template process NutrientLimitation(pp : PrimaryProducer, n : Nutrient) {}
template process MonodNutrientLim : NutrientLimitation {
  equations:
    pp.nutrientLim = n.conc / (n.conc + n.halfSaturation);
}
template process Monod2NutrientLim : NutrientLimitation {
    equations:
        pp.nutrientLim = n.conc * n.conc / (n.conc * n.conc + n.halfSaturation);
}
template process ExponentialNutrientLim : NutrientLimitation {
    equations:
        pp.nutrientLim = 1  exp(n.halfSaturation * n.conc);
}
template process Mortality(pp : PrimaryProducer) {
        consts:
            mortRate {range: <1, inf>};
        equations:
            td(pp.conc) = mortRate * phyto.conc;
}
template process Mixing(epiEntity : EcosystemEntity, hypoEntity : EcosystemEntity,
      env : Environment) {
    equations:
        td(epiEntity.conc) = env.flow * (epiEntity.conc  hypoEntity.conc),
        td(hypoEntity.conc) = env.flow * (hypoEntity.conc  epiEntity.conc);
}
template compartment Epilimnion {
    entities:
        EcosystemEntity;
    processes:
        Growth, NutrientLimitation;
}
template compartment Hypolimnion {
    entities:
        PrimaryProducer;
    processes:
        Mortality;
}
```

Example 3.10: A process-based model of a lake ecosystem based on the hierarchical version of the aquatic ecosystem library.

```
model lakeHierarchicalModel : AquaticHierarchicalLibrary;
compartment epi : Epilimnion {
    entity phyto : PrimaryProducer {
        vars:
            conc{role: endogenous; initial: 10},
            nutrientLim;
        consts:
            maxGrowthRate = 0.5;
    }
    entity phosphorus : Nutrient {
        vars:
            conc{role: endogenous; initial: 3};
        consts:
            halfSaturation = 0.02,
            alpha = 0.1;
    }
    process growth(phyto, [phosphorus]) : Growth {
        processes:
            phosphorusLim;
    }
    process phosphorusLim(phyto, phosphorus) : MonodLimitation{}
}
compartment hypo : Hypolimnion {
    entity phyto : PrimaryProducer {
        vars:
            conc{role: endogenous; initial: 10},
            nutrientLim;
        consts:
            maxGrowthRate = 0.5;
    }
    process mortality(phyto) : Mortality {
        consts:
            mortRate = 0.1;
    }
}
entity env : Environment{
    vars:
        flow{role: exogenous};
    consts:
        epiVolume = 6000000,
        hypoVolume = 20000000;
}
process mixing(epi.phyto, hypo.phyto, env) : Mixing {}
```

Inheritance is a notion that is well established in the theory of programming languages (Mitchell, 1996; Gordon, 1988). Object-oriented programming languages (Abadi and Cardelli, 1998; Gunter and Mitchell, 1994) use classes to structure the information used by the programs. In the case of single inheritance (as opposed to multiple inheritance), each class can inherit form one superclass. The properties, such as fields and methods, defined in the superclass are inherited to the subclass. In our proposed formalism, entity and process templates behave like classes in an object-oriented programming language and support the inheritance of properties in a analogous way.

4 Process-Based Modeling Formalism

In this chapter, we present a formalization of process-based models and background knowledge. The principal elements of the formalism were introduced in Chapter 3. We also introduced the rules that are used to build process-based models and libraries of background knowledge. For a library and a model to form a valid representation, they must satisfy a number of rules and constraints.

This chapter introduces a formal notation suitable for precise, mathematically sound, definitions of the key concepts of the process-based formalism. Process-based models and their components—entities, processes and compartments—as well as libraries and template entities, processes and compartments are then defined. The use of this formal notation and the provision of formal definitions simplify the task of expressing the properties of process-based models and libraries.

4.1 The Notation

We first present the notation for the basic concepts of process-based modeling, then introduce some auxiliary notation.

4.1.1 Notation for the Basic Concepts

Process-based libraries and their components are represented with the following sets of abstract concepts:

- $\mathcal{L}$, the set of all libraries,
- $\mathcal{TE}$, the set of all entity templates,
- $\mathcal{TP}$, the set of all process templates,
- $\mathcal{TK}$, the set of all compartment templates,
- $\mathcal{TV}$, the set of all variable templates,
- $\mathcal{TC}$, the set of all constant templates,
- $\mathcal{TQ}$, the set of all equation templates,

Process-based models and their components are represented with the following sets of abstract concepts:

- $\mathcal{M}$, the set of all models,
- $\mathcal{E}$, the set of all entities,
- $\mathcal{P}$, the set of all processes,
- $\mathcal{K}$, the set of all compartments,
- $\mathcal{V}$, the set of all variables,
- $\mathcal{C}$, the set of all constants,
- $\mathcal{Q}$, the set of all equations,

In the remainder of this section, we will give a formal mathematical definition for each concept from the lists above. The first subsection defines libraries and templates, while the second formalizes models and instances. Each concept is defined in term of its general properties and formal specification of its relation to other concepts. The notation allows us to precisely specify the template instantiation by providing formal definitions of the relations between templates and their instances.

4.1.2 Auxiliary Notation

Before we describe the elements of the formalism, we introduce some auxiliary notation.

Let A be a set. We use A^n to denote the Cartesian product $\underbrace{A \times \cdots \times A}_{\text{n times}}$, where $A^0 = \varnothing$, and $A^1 = A$;

Furthermore, we denote the closure of A with $A^* = \bigcup_{n=0}^{\infty} A^n$; it represents the set of all possible tuples of the elements of A.

We use $\mathscr{P}(A)$ to denote the power set of A, i.e., the set of all possible subsets of A, and $|A|$ to denote the cardinality (number of elements) of A.

Let $X = (x_1, x_2, \ldots, x_n)$ be a tuple. We denote the length of X by $|X| = n$.

Let $N_k = \{1, 2, \ldots, k\}$ be the set of the first k integers. The set of all permutations of N_k is denoted S_k.

Let $\mathbb{N}_a^b = \{n \in \mathbb{N}_0 | a \leq n \leq b\}$, $a, b \in \mathbb{N}_0$ be the closed integer interval between a and b.

Let $[\mathbb{N}] = \left\{\mathbb{N}_a^b | a, b \in \mathbb{N} \wedge a \leq b\right\}$ be the set of all possible integer intervals.

Let $R \subset A \times A$ be a relation in A. The transitive closure of R is denoted by $R^* = \bigcup_{n=1}^{\infty} R^n$, where $R^1 = R$, and for $n > 0$, $R^{n+1} = R \circ R^n$, where $\circ$ denotes the operation of composition of relations.

Let $f : A \to \mathscr{P}(A)$ be a function over a set A that maps an element of A into a subset of A. We define a sequence of functions: $f^1(x) = f(x)$, $f^2(x) = \left\{z | z \in f^1(y) \wedge y \in f^1(x)\right\}$, $f^3(x) = \left\{z | z \in f^1(y) \wedge y \in f^2(x)\right\}$, $\ldots$, $f^k(x) = \left\{z | z \in f^1(y) \wedge y \in f^{k-1}(x)\right\}$. We then denote the transitive closure of the function f as $f^* = \bigcup_{n=1}^{\infty} f^n$.

Let $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_m)$ be two tuples of length n and m respectively. We denote $X \subset Y$ if the first n elements in Y are the same as in X, i.e., $(\forall i \in \mathbb{N}_1^n) y_i = x_i$.

Let $f : A \to X$ and $g : B \to X$ be two functions such that $A \subset B$. Each function can be observed as a set of pairs $(a, f(a))$ and $(b, g(b))$. We denote $f \subset g$ when the set of pairs of f is a subset of the set of pairs of g, i.e., $\{(a, f(a)) | a \in A\} \subset \{(b, g(b)) | b \in B\}$.

4.2 Libraries and Templates

4.2.1 Libraries and Compartment Templates

A **library** is a named collection of entity, process and compartment templates.

Mathematically, each $L \in \mathcal{L}$ is fully specified with the following functions:

$$entities : \mathcal{L} \to \mathscr{P}(\mathcal{TE}) \tag{4.1a}$$
$$processes : \mathcal{L} \to \mathscr{P}(\mathcal{TP}) \tag{4.1b}$$
$$compartments : \mathcal{L} \to \mathscr{P}(\mathcal{TK}) \tag{4.1c}$$

In process-based formalism, a library is specified with the following syntax:

library *Name*;
template_defs

where *template_defs* is a sequence of entity, process and compartment definitions.

template_defs::= (*entity_template_def*|*process_template_def*|*compartment_template_def*)∗

Entity, process and compartment templates can be defined in an arbitrary order. Each template type is specified with a different keyword combination, therefore, no ambiguity can arise.

A **compartment template** is, just as a library, a named collection of entity, process and compartment templates.

Mathematically, each $TK \in \mathcal{TK}$ is fully specified with the following functions:

$$\begin{aligned} entities &: \mathcal{TK} \to \mathscr{P}(\mathcal{TE}) && (4.2a)\\ processes &: \mathcal{TK} \to \mathscr{P}(\mathcal{TP}) && (4.2b)\\ compartments &: \mathcal{TK} \to \mathscr{P}(\mathcal{TK}) && (4.2c) \end{aligned}$$

In the process-based formalism, a compartment template is specified with the following syntax:

template compartment *Name* {
  entities: *TE1*, *TE2*, ...;
  processes: *TP1*, *TP2*, ...;
  compartments: *TK1*, *TK2*, ...;
}

where *TE1*, *TE2*, ... are identifiers of existing entity templates, *TP1*, *TP2*, ... are identifiers of existing process templates, and *TK1*, *TK2*, ... are identifiers of existing compartment templates.

The order in which *entities*, *processes*, and *compartments* are specified does not have any influence. If the compartment template does not contain any entity templates, the *entities* part can be omitted. Similarly, *processes* and *compartments* can be omitted if there are no process or compartment templates, respectively.

4.2.2 Entity Templates

An **entity template** is a named collection of variable templates and constant templates. Mathematically, each $TE \in \mathcal{TE}$ is specified with the following functions:

$$\begin{aligned} variables &: \mathcal{TE} \to \mathscr{P}(\mathcal{TV}) && (4.3a)\\ constants &: \mathcal{TE} \to \mathscr{P}(\mathcal{TC}) && (4.3b) \end{aligned}$$

In the process-based formalism, an entity template is specified with the following syntax:

template entity *Name* [: *SuperEntity*] {
  vars:
    template_variable_def (,*template_variable_def*)∗;
  consts:
    template_constant_def (,*template_constant_def*)∗;
}

where *template_variable_def* is a variable template definition and *template_constant_def* is a constant template definition.

All variables and constants have a unique name by which they are identified. The order in which variables and constants are defined does not play any role, because they are identified by their name, not by their position. If there are no variables or constants in an entity template, we can omit the *vars* or *consts* sections, respectively.

The definition of an entity template contains an optional declaration of a *SuperEntity*. The role of the *SuperEntity* is discussed in detail later on. In short, the entity template inherits variable and constant templates from its *SuperEntity* template.

A **variable template** is a part of the specification of a variable that is contained in an entity template. The name of the variable template is unique within the specification of the entity template, but need not be unique across multiple entity templates. Therefore, when referring to a certain variable, it has to be clear to which entity template this variable belongs.

The variable template is characterized by its range, unit and aggregation function. Each $TV \in \mathcal{TV}$ is specified with the following functions:

$$range : \mathcal{TV} \to [a,b], \quad a,b \in \mathbb{R} \tag{4.4a}$$
$$unit : \mathcal{TV} \to A^*, \quad \text{where } A \text{ is an alphabet set} \tag{4.4b}$$
$$aggregation : \mathcal{TV} \to (\mathbb{R}^* \to \mathbb{R}) \tag{4.4c}$$

In the process-based formalism, a variable template is specified with the following syntax:

name {
  range: <*lower_bound*, *upper_bound*>;
  unit: *string_value*;
  aggregation: *aggregation_name*;
}

The range of a variable is the interval in which the values of the variable can fluctuate through time. The unit is a string designating the units in which this variable is measured. The range and unit are principally for annotation purposes.

The aggregation function is a function that maps any tuple of real numbers to a single real number. It is used for combining all influences from processes that have as an argument the entity containing the variable. The aggregation function (*agg*) has to be commutative so that the order in which the influences from individual processes are specified does not matter:

$$\Big(\forall n \in \mathbb{N}\Big)\Big(\forall (x_1,x_2,\ldots,x_n) \in \mathbb{N}^n\Big)\Big(\forall (i_1,i_2,\ldots,i_n) \in S_n\Big) agg(x_{i_1},x_{i_2},\ldots,x_{i_n}) = agg(x_1,x_2,\ldots,x_n) \tag{4.5}$$

The process-based formalism includes the following aggregation functions: *sum*, *product*, *average*, *minimum* and *maximum*.

A **constant template** is specified similarly to a variable template. The constant template is characterized by its range and unit. Each $TC \in \mathcal{TC}$ is specified with the following functions:

$$range : \mathcal{TC} \to [a,b], \quad a,b \in \mathbb{R} \tag{4.6a}$$
$$unit : \mathcal{TC} \to A^*, \quad \text{where } A \text{ is an alphabet set} \tag{4.6b}$$

In the process-based formalism, a constant template is specified with the following syntax:

name {
  range: <*lower_bound*, *upper_bound*>;
  unit: *string_value*;
}

4.2.3 Inheritance in Entity Templates

In each library, there is one special entity template denoted as TE_0 such that $variables(TE_0) = \varnothing$ and $constants(TE_0) = \varnothing$.

The $super^{\mathrm{E}}$ relation is defined between two entity templates, i.e., $super^{\mathrm{E}} \subset \mathcal{TE} \times \mathcal{TE}$ with the following property

$$\Big(\forall TE \in \mathcal{TE} \setminus \{TE_0\}\Big)\Big(\exists!\, TE_{sup} \in \mathcal{TE}\Big) super^{\mathrm{E}}(TE_{sup},TE) \tag{4.7}$$

In other words, every entity template TE other than TE_0 has a single super entity—TE_{sup}. We read $super^{\mathrm{E}}(A,B)$ as *A is a super entity template of B.*

The representation of an entity template in the process-based formalism described previously, contained a reference to *SuperEntity*. The *SuperEntity* is the super entity of the entity template being described.

The entity templates in a library form a taxonomy of entity templates in the form of a rooted tree. This tree has TE_0 as its root and links defined through the $super^{\mathrm{E}}$ relation. Each entity template other than TE_0 is connected to the root of the tree (TE_0) through exactly one path:

$$\begin{gathered}\left(\forall TE \in \mathcal{TE} \setminus TE_0\right)\left(\exists!\, k \in \mathbb{N}_0\right)\left(\exists!\,(a_1,a_2,\ldots,a_k) \in \mathcal{TE}^k\right) \\ super^{\mathrm{E}}(TE_0,a_1) \wedge super^{\mathrm{E}}(a_1,a_2) \wedge \cdots \wedge super^{\mathrm{E}}(a_k,TE)\end{gathered} \tag{4.8}$$

The inverse relation or $super^{\mathrm{E}}$ is the sub^{E} relation, formally defined as:

$$sub^{\mathrm{E}}(A,B) \iff super^{\mathrm{E}}(B,A) \tag{4.9}$$

We read $sub^{\mathrm{E}}(A,B)$ as *A is a sub entity template of B.* While every entity template other than TE_0 has a unique *super entity template*, it can have any number of, including zero, *sub entity templates.*

The $super^{\mathrm{E}}$ relation is used to define the $ancestor^{\mathrm{E}}$ relation which is a transitive closure of $super^{\mathrm{E}}$, i.e., $ancestor^{\mathrm{E}} = (super^{\mathrm{E}})^*$

We define $descendant^{\mathrm{E}}$ as an inverse relation of $ancestor^{\mathrm{E}}$ as:

$$descendant^{\mathrm{E}}(A,B) \iff ancestor^{\mathrm{E}}(B,A) \tag{4.10}$$

Each entity template inherits the variables and constants from its super entity template. The inheritance property is captured in the following requirement:

$$\begin{gathered}\left(\forall TE_1, TE_2 \in \mathcal{TE}\right) super^{\mathrm{E}}(TE_1,TE_2) \\ \implies variables(TE_1) \subseteq variables(TE_2) \wedge constants(TE_1) \subseteq constants(TE_2)\end{gathered} \tag{4.11}$$

In the process-based formalism, variables and constants are not redeclared in the subentity template. Moreover, an entity template cannot define any variable and constant with a name that exists in its superentity or any ancestor entity.

4.2.4 Process Templates

A process template relates entity templates. It provides details and quantifies this relation and provides constraints that the relation must satisfy. Each process template is characterized with its arguments, constants, equations, and nested processes.

A process template is defined with the following general syntax:

template process *Name* [(*Arguments*)] [: *SuperProcess*] {
    consts:
        template_constant_def (,*template_constant_def*)∗;
    equations:
        template_eq_def (,*template_eq_def*)∗;
    processes:
        nested_processes_list;
}

To illustrate the definition of a process template, we use the specification of the phytoplankton growth process introduced in Example 3.5:

template process *Growth*(*pp* : *PrimaryProducer*, *ns* : *Nutrient*<1,*inf*>) {
processes:
NutrientLimitation(*pp*, <*n*:*ns*>);
equations:
td(*pp.conc*) = *pp.maxGrowthRate* ∗ *pp.nutrientLim* ∗ *pp.conc*,
td(<*n*:*ns*>.*conc*) = *n.alpha* ∗ *pp.maxGrowthRate* ∗ *pp.nutrientLim* ∗ *pp.conc*;
}

In the following sections we describe each part of the definition of a process template.

Arguments of Process Templates The arguments of a process template specify which types of entities can be involved in the process. A process can contain zero, one or more arguments. Each argument of a process template serves as a constraint for the arguments of the process instances which will be created from this template.

The arguments are described with two mappings, one designating the allowed type of the argument and one designating the allowed argument cardinality:

$$arguments^{\mathrm{T}} : \boldsymbol{\mathcal{TP}} \to \boldsymbol{\mathcal{TE}}^*, \text{ \textit{types} of the arguments of each process template} \tag{4.12a}$$
$$arguments^{\mathrm{C}} : \boldsymbol{\mathcal{TP}} \to [\mathbb{N}]^*, \text{ \textit{cardinalities} of the arguments of each process template} \tag{4.12b}$$

The number of argument types and cardinality has to be consistent:

$$\left(\forall TP \in \boldsymbol{\mathcal{TP}}\right) |arguments^{\mathrm{T}}(TP)| = |arguments^{\mathrm{C}}(TP)| \tag{4.13}$$

In the process-based formalism, the arguments of a process template are defined with the following syntax:

(*argumentSpec1*, *argumentSpec2*, ..., *argumentSpecN*)

where

argumentSpec ::= *argumentName* : *argumentType*[<*minCard*, *maxCard*> | <*card*>]

and *argumentName* is an argument identifier used in the body of the process template and *argumentType* is the entity template specifying the type of the argument (the argument has to be an instance of that entity template). The last part of the specification designates the allowed cardinality. Cardinality is specified with lower and upper bound as the interval <*minCard, maxCard*>. However, if the lower and upper bound on the cardinality is the same, then the definition of cardinality can be shortened to <*card*>. If both the lower and upper cardinality of the argument is 1, then the cardinality declaration can be omitted, as 1 is the default cardinality for arguments. The definition of an argument then becomes:

argumentName : *argumentType*

In the *Growth* process template presented above, the argument specification is as follows:

(*pp* : *PrimaryProducer*, *ns* : *Nutrient*<1,*inf*>)

which means that the *Growth* process contains two arguments: the first named *pp* of type *PrimaryProducer* and implied cardinality 1, and the second argument named *ns* of type *Nutrient* and cardinality from 1 to infinity.

The *Growth* process has the following mappings for the *arguments*$^{\mathrm{T}}$ and *arguments*$^{\mathrm{C}}$ functions associated to it:

$$arguments^{\mathrm{T}} : (\mathrm{PrimaryProducer}, \mathrm{Nutrient}) \tag{4.14a}$$
$$arguments^{\mathrm{C}} : (\{1\}, \mathbb{N}) \tag{4.14b}$$

Constant Templates Process templates contain constants which are concepts of the same type as in entity templates.

$$\mathit{constants} : \mathcal{TP} \rightarrow \mathscr{P}(\mathcal{TC}) \tag{4.15}$$

Which means that the *constants* mapping is defined for both $\mathcal{TE}$ and $\mathcal{TP}$.

Variables and constants in process templates The arguments of a process template influence the equations and nested processes of the process template. In order to facilitate the definitions of equations and nested processes, we define two auxiliary mappings: $\mathit{arg}^{\mathrm{TV}}$ and $\mathit{arg}^{\mathrm{TC}}$.

For each process template, $\mathit{arg}^{\mathrm{TV}}$ specifies the set of all template variables from the arguments of the process. In order to identify them, we need to know the index of the argument and the template variable:

$$\mathit{arg}^{\mathrm{TV}}(TP) = \left\{ (i, TV) \in \mathbb{N} \times \mathcal{TV} \middle| i \in \mathbb{N}_1^{|\mathit{arguments}^{\mathrm{T}}(TP)|} \wedge TV \in \mathit{variables}\left(\mathit{arguments}^{\mathrm{T}}(TP)_i\right) \right\} \tag{4.16}$$

For each process template, $\mathit{arg}^{\mathrm{TC}}$ specifies the set of all template constants from the arguments of the process, defined in a similar manner as $\mathit{arg}^{\mathrm{TC}}$:

$$\mathit{arg}^{\mathrm{TC}}(TP) = \left\{ (i, TC) \in \mathbb{N} \times \mathcal{TC} \middle| i \in \mathbb{N}_1^{|\mathit{arguments}^{\mathrm{T}}(TP)|} \wedge TC \in \mathit{constants}\left(\mathit{arguments}^{\mathrm{T}}(TP)_i\right) \right\} \tag{4.17}$$

In the case of the *Growth* process (assuming that *Nutrient* and *PrimaryProducer* are defined as in Example 3.5), the $\mathit{arg}^{\mathrm{TV}}$ and $\mathit{arg}^{\mathrm{TC}}$ functions yield the following values:

$$\mathit{arg}^{\mathrm{TV}} : \{(1, \mathrm{conc}), (1, \mathrm{nutrientLim}), (2, \mathrm{conc})\} \tag{4.18a}$$
$$\mathit{arg}^{\mathrm{TC}} : \{(1, \mathrm{maxGrowthRate}), (2, \mathrm{halfSaturation}), (2, \mathrm{alpha})\} \tag{4.18b}$$

Equation Templates Process templates contain tuples of equation templates:

$$\mathit{equations} : \mathcal{TP} \rightarrow \mathcal{TQ}^* \tag{4.19}$$

Equations are described through five mappings defined over the set of equation templates. In the following definitions, TP is the process template that contains the equation and k can be any non-negative integer, the value of which is the same across all definitions.

$$\mathit{type} : \mathcal{TQ} \rightarrow \{\mathrm{algebraic}, \mathrm{differential}\} \tag{4.20a}$$
$$\mathit{lhs} : \mathcal{TQ} \rightarrow \mathit{arg}^{\mathrm{TV}}(TP) \tag{4.20b}$$
$$\mathit{function} : \mathcal{TQ} \rightarrow (\mathbb{R}^k \rightarrow \mathbb{R}) \tag{4.20c}$$
$$\mathit{inputs} : \mathcal{TQ} \rightarrow \left(\mathbb{N}_1^k \rightarrow (\mathit{arg}^{\mathrm{TV}}(TP) \cup \mathit{arg}^{\mathrm{TC}}(TP) \cup \mathit{constants}(TP))\right) \tag{4.20d}$$
$$\mathit{iterated} : \mathcal{TQ} \rightarrow \mathbb{N}_0^k \tag{4.20e}$$

Each equation template can refer to an algebraic or a differential equation, which is specified with the *type* function. The left-hand side of the equation consists of a template variable from an argument of the template process, i.e., an element of $\mathit{arg}^{\mathrm{TV}}$, which is specified with the *lhs* function. On the right-hand side of the equation, we distinguish two main components: the mathematical functional form of the right-hand side, and the variables and constants that appear on the right-hand side. The functional form of the right-hand side is specified with the mapping *function*. The functional form can be any mathematical function. In the process based formalism, we support functions that are expressed as formulas containing the following operators and functions: unary negation ($-$), addition ($+$), subtraction ($-$), multiplication ($*$), division ($/$), sine (sin), cosine (cos), signum (*sign*), power (*pow*), minimum (*min*), maximum (*max*), exponential (*exp*), natural logarithm (*log*) and common logarithm (*log10*). The *function* mapping thus specifies a mathematical function of k inputs. Each of the k inputs can be a variable or a constant template from the arguments of the process or a local constant template from the process. The ***inputs*** mapping maps each input positions to a variable or constant template that is used as an input at the corresponding position.

If we look at the first equation of the *Growth* process (denote it as q):

td(*pp.conc*) = *pp.maxGrowthRate* ∗ *pp.nutrientLim* ∗ *pp.conc*,

then this is a differential equation—$type(q) = \text{differential}$, it has as a left-hand side the *conc* variable of the first argument (pp)—$lhs(q) = (1, \text{conc})$, the right-hand side has three inputs—$k = 3$, the functional form of the right-hand side is a multiplication of three numbers—$x_1 * x_2 * x_3$, and the mappings of inputs to variables and constants is given with the following scheme:

$$\begin{pmatrix} 1 & 2 & 3 \\ (1, \text{maxGrowthRate}) & (1, \text{nutrientLim}) & (1, \text{conc}) \end{pmatrix} \tag{4.21}$$

Iterators We mentioned that an instance process argument can contain a set of instance entities. The number of elements in the set depends on the system we are modeling. Within the library, we allow sets of entities as arguments with the specification of cardinality of the suitable template process argument. Sometimes, a need arises to make statements that must hold for every instance in the argument set, without knowing the number of instances in the set. To this end, process-based modeling formalism employs *iterators*. They allow us to make a statement *for every* entity in an argument set, but also to make multiple instances of a given template, where each instance corresponds to one of the elements from the argument set. We can use iterators with template equations and nested processes.

The equation template modeling the dynamic change of the nutrient concentration is an example of an iterated template. The iterator is specified with the construct <*n:ns*>, which denotes that the process template specifies one equation for each element n of the set of nutrients ns (we say that the equation is iterated over ns).

To illustrate the use of iterators, we look at the equation for nutrient concentration in the *Growth* process:

td(<*n*:*ns*>.*conc*) = *n.alpha* ∗ *pp.maxGrowthRate* ∗ *pp.nutrientLim* ∗ *pp.conc*;

The mappings for this equation are defined with:

$$type : \text{differential} \tag{4.22a}$$

$$lhs : (2, \text{conc}) \tag{4.22b}$$

$$function : -x_1 * x_2 * x_3 * x_4 \tag{4.22c}$$

$$inputs : \begin{pmatrix} 1 & 2 & 3 & 4 \\ (2, \text{alpha}) & (1, \text{maxGrowthRate}) & (1, \text{nutrientLim}) & (1, \text{conc}) \end{pmatrix} \tag{4.22d}$$

The definition of the *growthPhyto* process instance from Example 3.6 relies on *Growth* process template:

process *growthPhyto*(*phyto*, [*phosphorus*, *nitrogen*]) : *Growth* {
  processes:
    phosphorusLim, *nitrogenLim*;
}

The *ns* argument of the *Growth* process template is populated with the two-element set [phosphorus, nitrogen] in the instance process *growthPhyto*. The template equation for nutrient concentration:

td(<*n*:*ns*>.*conc*) = *n.alpha* ∗ *pp.maxGrowthRate* ∗ *pp.nutrientLim* ∗ *pp.conc*;

then transforms to two equations:

td(*phosphorus.conc*) = *phosphorus.alpha* ∗ *pp.maxGrowthRate* ∗ *pp.nutrientLim* ∗ *pp.conc*;
td(*nitrogen.conc*) = *nitrogen.alpha* ∗ *pp.maxGrowthRate* ∗ *pp.nutrientLim* ∗ *pp.conc*;

Note that each template equation can include at most one iterator.

The variables or constants used in an equation have to come from arguments of the process with cardinality one. Otherwise, we would not know which variable or constant should be used in the equation. The exception is that the variables and constants can come from an argument with an arbitrary cardinality, if this argument is used as an iterator for the equation. In this case, for each instance of the argument there is one equation with the suitable variables and constants.

Mathematically, the following property holds for equation templates (where n is the number of inputs to the equation):

$$\begin{aligned}\left(\forall TQ \in \mathcal{TQ}\right)\left(\forall i \in \mathbb{N}_1^n\right) inputs(TQ)(i) = (k, T) \in arg^{\mathrm{TV}}(TP) \cup arg^{\mathrm{TC}}(TP) \\ \implies arguments^{\mathrm{C}}(TQ)_k = \{1\} \vee iterated(TQ) = k\end{aligned} \tag{4.23}$$

Nested Processes Nested processes are used to decompose a large and complex process into several smaller processes. A process can contain an arbitrary number of nested processes. Each nested process can in turn contain its own nested processes. The processes which are not nested in any other process are called *top-level processes*.

All process templates are defined at the library level. When specifying nested processes, we reference existing process templates. In addition to the process template, we also provide mappings of the arguments of the nested process to the arguments of the process in which the nesting occurs.

To facilitate the definition of the mapping of arguments between the nested and containing process, we define a mapping $nested^{\mathrm{ARG}} : \mathcal{TP} \to \mathscr{P}(\mathbb{N}^2)$, which for each process templates defines an index set that gives unique identifiers to all argument positions of nested processes.

Let $TP \in \mathcal{TP}$ be a process template. Then:

$$nested^{\mathrm{ARG}}(TP) = \left\{(i,j) \in \mathbb{N}^2 \middle| i \in \mathbb{N}_1^{|processes^{\mathrm{T}}(TP)|}, j \in \mathbb{N}_1^{|arguments^{\mathrm{T}}(processes^{\mathrm{T}}(TP)_i)|}\right\} \tag{4.24}$$

In other words, it is a set of pairs, where the first element is the index of the nested process and the second element is the index of the argument within the nested process.

Nested processes are described through the following mappings:

$$processes^{\mathrm{T}} : \mathcal{TP} \to \mathcal{TP}^*, \text{ types of the nested processes} \tag{4.25a}$$

$$\begin{aligned}&processes^{\mathrm{A}} \\ &\quad : \mathcal{TP} \to \left(nested^{\mathrm{ARG}}(TP) \to \mathbb{N}_1^{|arguments^{\mathrm{T}}(TP)|}\right), \\ &\quad \text{mapping between the arguments of the nested processes and the arguments of } TP\end{aligned} \tag{4.25b}$$

$$\begin{aligned}processes^{\mathrm{I}} : \mathcal{TP} \to \left(\mathbb{N}_1^{|processes^{\mathrm{T}}(TP)|} \to \left(\mathbb{N}_1^{|arguments^{\mathrm{T}}(TP)|} \cup \{0\}\right)\right), \\ \text{iterators of the nested processes}\end{aligned} \tag{4.25c}$$

We illustrate nested processes with the *Growth* process template from Example 3.9:

template process *Growth*(*pp* : *PrimaryProducer*, *ns* : *Nutrient*<1, *inf*>) {
  processes:
    NutrientLimitation(*pp*, <*n*:*ns*>);
  equations:
    ...
}

The *Growth* process template contains nested processes of type *NutrientLimitation.* The first argument to *NutrientLimitation* is the same as the first argument to *Growth*, designated as *pp*. The second argument of *NutrientLimitation* is iterated for all entities in *ns*, which means that there will be as many nested processes of type *NutrientLimitation* as entity instance in the *ns* argument in the process instance crated from this template.

Since the arguments of each nested process are taken from the arguments of the containing process, they must match in type and cardinality.

We can illustrate this with the *NutrientLimitation* process template from Example 3.9:

template process *NutrientLimitation*(*pp* : *PrimaryProducer*, *n* : *Nutrient*) {}

We can see that the first argument of the *Growth* process and the first argument of the *NutrientLimitation* process are both of type *PrimaryProducer* and both have cardinality one. The second argument of *Growth* and the second argument of *NutrientLimitation* are of the same type—Nutrient, but have different cardinalities. However, the reference of *NutrientLimitation* as a nested process of *Growth*, iterates across all instances in the second argument, designated with the iterator <*n:ns*>.

Mathematically, this requirement is formalized as:

$$\left(\forall TP \in \mathcal{TP}\right)\left(\forall i \in \mathbb{N}_1^{|processes^{\mathrm{T}}(TP)|}\right)\widehat{TP} = processes^{\mathrm{T}}(TP)_i \implies \left(\forall j \in \mathbb{N}_1^{\left|arguments^{\mathrm{T}}(\widehat{TP})\right|}\right)$$
$$\Big(k = processes^{\mathrm{A}}(TP)(i,j) \wedge \Big(descendant^{\mathrm{E}}(arguments^{\mathrm{T}}(\widehat{TP})_j, arguments^{\mathrm{T}}(TP)_k)$$
$$\wedge\ (arguments^{\mathrm{C}}(TP)_k \subseteq arguments^{\mathrm{C}}(\widehat{TP})_j \vee processes^{\mathrm{I}}(TP)_i = k)\Big)\Big) \tag{4.26}$$

Cyclic Nesting of Process Templates The process-based formalism does not allow for cyclic nesting of process templates. The nesting of process templates can therefore be regarded as a taxonomy.

In order to formally express this property, we define an auxiliary mapping $\overline{processes}$ which provides the same information as $processes^{\mathrm{T}}$, but in a form of a set instead of a tuple:

$$\left(\forall TP \in \mathcal{TP}\right)\overline{processes}(TP) = \bigcup_{i=1}^{|processes^{\mathrm{T}}(TP)|} processes^{\mathrm{T}}(TP)_i \tag{4.27}$$

Its transitive closure $\overline{processes}^*$ gives the set of all process templates that can be nested in the given process template at any level of nesting. In order to prevent cyclic nesting, we need to ensure that whenever a process template appears as a nested process of another process template (at any level of nesting), the second process template does not appear as a nested process of the first process:

$$\left(\forall TP_1, TP_2 \in \mathcal{TP}\right)TP_2 \in \overline{processes}^*(TP_1) \implies TP_1 \notin \overline{processes}^*(TP_2) \tag{4.28}$$

4.2.5 Inheritance in Process Templates

Similarly to the entity templates, process templates are also organized in an inheritance hierarchy. In each library, there is one special process template denoted as TP_0 such that $arguments(TP_0) = \varnothing$, $constants(TP_0) = \varnothing$, $processes(TP_0) = \varnothing$ and $equations(TP_0) = \varnothing$.

The definition of the $super^{\mathrm{P}}$ relation extends to process templates. It is defined between two process templates, i.e., $super^{\mathrm{P}} \subset \mathcal{TP} \times \mathcal{TP}$ with the following property:

$$\left(\forall TP \in \mathcal{TP} \setminus \{TP_0\}\right)\left(\exists!\, TP_{sup} \in \mathcal{TP}\right)super^{\mathrm{P}}(TP_{sup}, TP) \tag{4.29}$$

Every process template TP other than TP_0 has a single super process—TP_{sup}. When A and B are process templates, we read $super^{\mathrm{P}}(A,B)$ as *A is a super process template of B.* The definition of a process template in the process-based formalism contained an optional reference to *SuperProcess.* The *SuperProcess* is the super process of the process template being described. If the specification of super process is omitted from the definition of a process template, then it is assumed that TP_0 is its super process.

The process templates in a library form a taxonomy of process templates in the form of a rooted tree. This tree has TP_0 as its root and links defined through the $super^{\mathrm{P}}$ relation. Each process template other than TP_0 is connected to the root of the tree through exactly one path:

$$\begin{gathered}\Big(\forall TP \in \boldsymbol{\mathcal{TP}} \setminus TP_0\Big)\Big(\exists!\, k \in \mathbb{N}_0\Big)\Big(\exists!\,(a_1,a_2,\ldots,a_k) \in \boldsymbol{\mathcal{TP}}^k\Big)\\ super^{\mathrm{P}}(TP_0,a_1) \wedge super^{\mathrm{P}}(a_1,a_2) \wedge \cdots \wedge super^{\mathrm{P}}(a_k,TP)\end{gathered} \tag{4.30}$$

We extend the notions of sub^{E}, $ancestor^{\mathrm{E}}$ and $descendant^{\mathrm{E}}$ relations to process templates. They are defined analogously as for entity templates.

$$sub^{\mathrm{P}}(A,B) \iff super^{\mathrm{P}}(B,A) \tag{4.31a}$$

$$ancestor^{\mathrm{P}} = (super^{\mathrm{P}})^{*} \tag{4.31b}$$

$$descendant^{\mathrm{P}}(A,B) \iff ancestor^{\mathrm{P}}(B,A) \tag{4.31c}$$

Each process template inherits the arguments, constants, equations and nested processes from its super process template. The inheritance property is captured in the following requirement:

$$\begin{aligned}\Big(\forall TP_1,TP_2 \in \boldsymbol{\mathcal{TP}}\Big)&super^{\mathrm{P}}(TP_1,TP_2) \implies\\ &arguments(TP_1) \subseteq arguments(TP_2) \wedge\\ &constants(TP_1) \subseteq constants(TP_2) \wedge\\ &equations(TP_1) \subseteq equations(TP_2) \wedge\\ &processes(TP_1) \subseteq processes(TP_2)\end{aligned} \tag{4.32}$$

4.3 Models and Instances

Models represent named collections of instances. They contain entity and process instances organized in compartment instances. Entity instances, in turn, contain variable and constant instances. Process instances, on the other hand, contain constant and equation instances. Each instance in a model has a template which serves as the basis for the creation of the object. It is specified with the function $template(.)$. This function maps the object to a corresponding object from the library.

4.3.1 Models

A model is a collection of entities, processes and compartments. Each $M \in \boldsymbol{\mathcal{M}}$ is described with the following functions:

$$template : \boldsymbol{\mathcal{M}} \to \boldsymbol{\mathcal{L}} \tag{4.33a}$$

$$entities : \boldsymbol{\mathcal{M}} \to \mathscr{P}(\boldsymbol{\mathcal{E}}) \tag{4.33b}$$

$$processes : \boldsymbol{\mathcal{M}} \to \mathscr{P}(\boldsymbol{\mathcal{P}}) \tag{4.33c}$$

$$compartments : \boldsymbol{\mathcal{M}} \to \mathscr{P}(\boldsymbol{\mathcal{K}}) \tag{4.33d}$$

In the process-based formalism, a model is defined with the following syntax:

model *modelName* : *LibraryName*;
instance_defs

where *instance_defs* is a sequence of entity, process and compartment instance definitions.

instance_defs::= (*entity_instance_def* | *process_instance_def* | *compartment_instance_def*)∗

Entity, process and compartment instances can be defined in an arbitrary order. Each instance type is specified with a different keyword, therefore, no ambiguity can arise.

A model has a library object as a template. Moreover, the entity, process and compartment instances contained in the model have their templates in the library, which serves as a template for the model. Mathematically, the following properties must hold:

$$\Big(\forall E \in entities(M)\Big)\Big(\exists TE \in entities(template(M))\Big) template(E) = TE \tag{4.34a}$$

$$\Big(\forall P \in processes(M)\Big)\Big(\exists TP \in processes(template(M))\Big) template(P) = TP \tag{4.34b}$$

$$\Big(\forall K \in compartments(M)\Big)\Big(\exists TK \in compartments(template(M))\Big) template(K) = TK \tag{4.34c}$$

4.3.2 Compartments

A compartment is very similar to a model and acts like a mini-model. It also consists of entities, processes and nested compartments.

$$template : \boldsymbol{\mathcal{K}} \to \boldsymbol{\mathcal{TK}} \tag{4.35a}$$

$$entities : \boldsymbol{\mathcal{K}} \to \mathscr{P}(\boldsymbol{\mathcal{E}}) \tag{4.35b}$$

$$processes : \boldsymbol{\mathcal{K}} \to \mathscr{P}(\boldsymbol{\mathcal{P}}) \tag{4.35c}$$

$$compartments : \boldsymbol{\mathcal{K}} \to \mathscr{P}(\boldsymbol{\mathcal{K}}) \tag{4.35d}$$

In the process-based formalism, a compartment is defined with the following syntax:

compartment *compartmentName* : *TemplateCompartment* {
  instance_defs
}

where *compartmentName* is the name of the compartment, *TemplateCompartment* is the name of the compartment template and *instance_defs* has the same meaning as above.

Compartments can be nested in other compartments, forming a taxonomy of nested compartments. The topmost compartments are at the level of the model. The set of models and compartments $\boldsymbol{\mathcal{M}} \cup \boldsymbol{\mathcal{K}}$ together with the *compartments* function form a set of rooted trees (a forest) where the roots of the trees are models.

4.3.3 Identifiers of Instances

Two entities that appear in the same model as top-level entities or appear in the same compartment have to have different names, because their names are used as unique identifiers within the model or compartment. The same rule applies to processes and compartments. Entities (processes and compartments) that are contained in different compartments can have the same name, because they can be identified by their *qualified name*.

Identification of instances in the model works in a similar fashion to identification of files on a file system. Each instance has its *short name* which is the name with which it is defined and consists of a single identifier. This is equivalent to a filename of a file. In addition to a short name, each instance can be identified by its *fully qualified name* which consists of the simple names of all concepts in which the variable is nested separated by a dot.

4.3.4 Entities

An entity is a named collection of variables and constants. Mathematically, each $E \in \mathcal{E}$ is specified with the following mappings:

$$template : \mathcal{E} \to \mathcal{TE} \tag{4.36a}$$
$$variables : \mathcal{E} \to \mathcal{V} \tag{4.36b}$$
$$constants : \mathcal{E} \to \mathcal{C} \tag{4.36c}$$

In the process-based formalism, an entity is defined with the following syntax:

```
entity entityName : TemplateEntity {
    vars:
        instance_variable_def (,instance_variable_def)∗;
    consts:
        instance_constant_def (,instance_constant_def)∗;
}
```

where *instance_variable_def* is a variable instance definition and *instance_constant_def* is a constant instance definition.

Each entity instance must contain the same variables and constants as its template. The order in which they are provided, however, is not important and can differ in the entity instance and entity template. There is a one-to-one correspondence between $variables(E)$ and $variables(template(E))$ on one hand and $constants(E)$ and $constants(template(E))$ on the other:

$$\Big(\forall V \in variables(E)\Big)\Big(\exists! \, TV \in variables(template(E))\Big) template(V) = TV \tag{4.37a}$$
$$\Big(\forall TV \in variables(template(E))\Big)\Big(\exists! \, V \in variables(E)\Big) template(V) = TV \tag{4.37b}$$

$$\Big(\forall C \in constants(E)\Big)\Big(\exists! \, TC \in constants(template(E))\Big) template(C) = TC \tag{4.38a}$$
$$\Big(\forall TC \in constants(template(E))\Big)\Big(\exists! \, C \in constants(E)\Big) template(C) = TC \tag{4.38b}$$

Variables

A variable is characterized with its initial value, role, value, range, unit, and aggregation function.

$$template : \mathcal{V} \to \mathcal{TV} \tag{4.39a}$$
$$range : \mathcal{V} \to [a,b], \quad a,b \in \mathbb{R} \tag{4.39b}$$
$$unit : \mathcal{V} \to A^*, \quad \text{where } A \text{ is an alphabet set} \tag{4.39c}$$
$$aggregation : \mathcal{V} \to (\mathbb{R}^* \to \mathbb{R}) \tag{4.39d}$$
$$role : \mathcal{V} \to \{endogenous, exogenous\} \tag{4.39e}$$
$$initial : \mathcal{V} \to \mathbb{R} \tag{4.39f}$$
$$value : \mathcal{V} \to (\mathbb{R} \to \mathbb{R}) \tag{4.39g}$$

The range, unit, and aggregation function of each variable are the same as in its template.

$$range(V) = range(template(V)) \tag{4.40a}$$
$$unit(V) = unit(template(V)) \tag{4.40b}$$
$$aggregation(V) = aggregation(template(V)) \tag{4.40c}$$

The *role* of the variable can be either *exogenous* or *endogenous* (Figure 4.1). Exogenous variables are input variables that are used as forcing influences to the system. They are not modeled within the system, their behavior through time comes from external measurements.

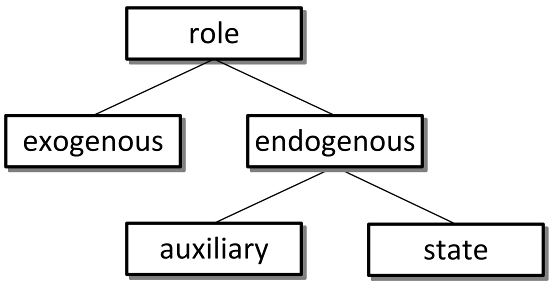


Figure 4.1: Taxonomy of roles of model variables

Endogenous variables, on the other hand, are modeled within the system. Each endogenous variable is assigned an equation (possible through combining several equation fragments), with which its value is computed.

Endogenous variables can be further classified as *auxiliary* or *state*. State variables are influenced by differential equations, whereas auxiliary variables are influenced by algebraic equations. A variable cannot be influenced by both algebraic and differential equations.

These requirements can be formally expressed with the following predicates:

$$auxiliary(v) = \text{true} \iff \Big(\forall q \in \mathcal{Q}\Big)\Big(lhs(q) = v \implies type(q) = \text{algebraic}\Big) \quad (4.41a)$$

$$state(v) = \text{true} \iff \Big(\forall q \in \mathcal{Q}\Big)\Big(lhs(q) = v \implies type(q) = \text{differential}\Big) \quad (4.41b)$$

State variables have an *initial value*, which is the value of the variable in the first time point. The value of each variable is a function of time, represented by the *value* mapping. The value of the variable is not included in the specification of variable instance in the process-based formalism. The value is assigned depending on the role of the variable. Exogenous variables acquire their values from measurements specified in an external data set. On the other hand, the values of endogenous variables are to be computed using the corresponding equations. In the case of auxiliary variables, the values are computed simply by employing the equations directly. In the case of state variables, differential equations are being envoked through numerical simulation.

Constants

A constant is characterized with its value, range, and unit.

$$template : \mathcal{C} \to \mathcal{TC} \quad (4.42a)$$

$$value : \mathcal{C} \to \mathbb{R} \quad (4.42b)$$

$$range : \mathcal{C} \to [a,b], \quad a,b \in \mathbb{R} \quad (4.42c)$$

$$unit : \mathcal{C} \to A^*, \quad \text{where } A \text{ is an alphabet set} \quad (4.42d)$$

The range and unit of each constant are the same as in its template.

$$range(C) = range(template(C)) \quad (4.43a)$$

$$unit(C) = unit(template(C)) \quad (4.43b)$$

The value of the constant is its only property which is specified when instantiating the constant. Therefore, it makes sense to make the assignment of a value to a constant straightforward, by simply assigning the value to the constant name itself with the syntax: $constName = realNumber$.

4.3.5 Processes

A process instance relates entity instances. A process is characterized with its arguments, constants, equations and nested processes.

A process instance is defined with the following syntax:

process *processName* (*Arguments*) : *TemplateProcess* {
  consts:
    instance_constant_def (,*instance_constant_def*)∗;
  processes:
    nested_processes_list;
}

Mathematically, for each process instance the following mappings are defined:

$$template : \mathcal{P} \to \mathcal{TP} \tag{4.44a}$$
$$arguments : \mathcal{P} \to \mathscr{P}(\mathcal{E})^* \tag{4.44b}$$
$$constants : \mathcal{P} \to \mathcal{C} \tag{4.44c}$$
$$equations : \mathcal{P} \to \mathscr{P}(\mathcal{Q})^* \tag{4.44d}$$
$$processes : \mathcal{P} \to \mathscr{P}(\mathcal{P})^* \tag{4.44e}$$

Each argument of a process instance is a set of entity instances. In the process-based formalism, sets are denoted with square brackets as delimiters. For example, *[phosphorus, nitrogen]* denotes a two-element set consisting of the entities *phosphorus* and *nitrogen*. For simplicity, single-element sets can be written without the square brackets.

The number of arguments of a process must correspond to the number of arguments of its process template. Furthermore, the type and number of the entities in each argument must correspond to the type and cardinality of the argument of the process template.

The following properties must hold for the arguments of any process P:

$$|arguments(P)| = |arguments^{\mathrm{T}}(template(P))| \tag{4.45a}$$

$$\left(\forall i \in \mathbb{N}_1^{|arguments(P)|}\right)\left(\forall E \in arguments(P)_i\right) \\ descendant^{\mathrm{E}}\left(template(E), arguments^{\mathrm{T}}(template(P))_i\right) \tag{4.45b}$$

$$\left(\forall i \in \mathbb{N}_1^{|arguments(P)|}\right) |arguments(P)_i| \in arguments^{\mathrm{C}}(template(P))_i \tag{4.45c}$$

Constants in processes are specified in the same way as constants in entities. Additionally, there is a one-to-one mapping between the constant instances in the process instance and the constant templates in the process template.

$$\left(\forall C \ \in constants(P)\right)\left(\exists! TC \in constants(template(P))\right) template(C) = TC \tag{4.46a}$$

$$\left(\forall TC \ \in constants(template(P))\right)\left(\exists! C \in constants(P)\right) template(C) = TC \tag{4.46b}$$

Equations

Each equation instance is defined with its function, type, and left-hand side and input mappings.

$$type : \mathcal{Q} \to \{\text{algebraic}, \text{differential}\}, \text{ whether the equation is algebraic or differential} \tag{4.47a}$$
$$lhs : \mathcal{Q} \to \mathcal{V}, \text{ the variable that is the left-hand side of the equation} \tag{4.47b}$$
$$function : \mathcal{Q} \to (\mathbb{R}^n \to \mathbb{R}), \text{ the functional form of the right-hand side of the equation} \tag{4.47c}$$
$$inputs : \mathcal{Q} \to (\mathbb{N}_1^n \to (\mathcal{V} \cup \mathcal{C})), \tag{4.47d}$$

which variables and constants are used in the equation and the exact positions on which they are used

$$\mathit{iterated} : \boldsymbol{\mathcal{Q}} \to \boldsymbol{\mathcal{E}} \cup \{\varnothing\}, \text{ the entity for which this equation is iterated, if any} \tag{4.47e}$$

The type and functional form of each equation are the same as for its template equation.

$$\mathit{type}(Q) = \mathit{type}(\mathit{template}(Q)) \tag{4.48a}$$
$$\mathit{function}(Q) = \mathit{function}(\mathit{template}(Q)) \tag{4.48b}$$

Equation instances have special treatment in the process-based formalism because they are the only concepts that are not explicitly defined. There is no way to define an equation instance in a model. Equation instances are implicitly instantiated whenever a process instance is instantiated. The equation instance in uniquely determined with the equation template defined in the process template and the arguments of the process. Equation instances, however, like all other concepts in the formalism can be mathematically specified with mappings and set of propositions which must hold.

Each equation template corresponds to a set of equation instances. Each equation instance in the set must have the corresponding equation template. The following properties hold for the equations of any process P:

$$|\mathit{equations}(P)| = |\mathit{equations}(\mathit{template}(P))| \tag{4.49a}$$
$$\left(\forall i \in \mathbb{N}_1^{|\mathit{equations}(P)|}\right)\left(\forall Q \in \mathit{equations}(P)_i\right)\mathit{template}(Q) = \mathit{equations}(\mathit{template}(P))_i \tag{4.49b}$$

When we talked about process templates, we defined the mappings $\mathit{arg}^{\mathrm{TV}}$ and $\mathit{arg}^{\mathrm{TC}}$ which for a given process template provide all variable and constant templates, respectively, that belong to the arguments of that process template. Here, we define a function $\overline{\mathit{arg}}$ that maps variable and constant templates from the arguments of a process template to variable and constant instances from the arguments of a process instance.

$$\overline{\mathit{arg}} : \mathit{arg}^{\mathrm{TV}}(\mathit{template}(P)) \cup \mathit{arg}^{\mathrm{TC}}(\mathit{template}(P)) \cup \mathit{constants}(\mathit{template}(P)) \to \boldsymbol{\mathcal{V}} \cup \boldsymbol{\mathcal{C}} \tag{4.50a}$$

$$\left(\forall (i, TV) \in \mathit{arg}^{\mathrm{TV}}(\mathit{template}(P))\right)$$
$$\overline{\mathit{arg}}((i,TV)) = \left\{V \in \boldsymbol{\mathcal{V}} \middle| (\exists E \in \mathit{arguments}(P)_i) V \in \mathit{variables}(E) \land \mathit{template}(V) = TV\right\} \tag{4.50b}$$

$$\left(\forall (i, TC) \in \mathit{arg}^{\mathrm{TC}}(\mathit{template}(P))\right)$$
$$\overline{\mathit{arg}}((i,TC)) = \left\{C \in \boldsymbol{\mathcal{C}} \middle| (\exists E \in \mathit{arguments}(P)_i) C \in \mathit{constants}(E) \land \mathit{template}(C) = TC\right\} \tag{4.50c}$$

$$\left(\forall TC \in \mathit{constants}(\mathit{template}(P))\right)\overline{\mathit{arg}}(TC) = \left\{C \in \mathit{constants}(P) \middle| \mathit{template}(C) = TC\right\} \tag{4.50d}$$

All equation templates that are not iterated ($\mathit{iterated}(TQ) = 0$) use only arguments with cardinality one. That means that the subsets of $\mathit{arg}^{\mathrm{TV}}(TP)$ and $\mathit{arg}^{\mathrm{TC}}(TP)$ that are used in the equations map with $\overline{\mathit{arg}}$ to single element sets. The *input* mapping for these equations is then defined as a composition of the *input* mapping of its equation template and the $\overline{\mathit{arg}}$ mapping.

$$\mathit{iterated}(\mathit{template}(Q)) = 0 \implies \mathit{inputs}(Q)(i) \in \overline{\mathit{arg}}(\mathit{inputs}(\mathit{template}(Q))(i)) \tag{4.51}$$

The iterated argument of an equation that is iterated can contain arbitrary number of entities as long as this satisfies the constraint of the cardinality of the argument. For each of this entities there is a one equation that has that entity as its *iterated* mapping.

$$\begin{gathered}\left(\forall P \in \boldsymbol{\mathcal{P}}\right)\left(\forall i \in \mathbb{N}_1^{|\mathit{equations}(\mathit{template}(P))|}\right)(TQ = \mathit{equations}(\mathit{template}(P)_i)) \implies \\ \left(\forall E \in \mathit{arguments}(P)_{\mathit{iterator}(TQ)}\right)\left(\exists Q \in \mathit{equations}(P)_i\right)\mathit{iterated}(Q) = E\end{gathered} \tag{4.52}$$

All template inputs (variable and constants) in an equation that come from the argument which is used as an iterator must come from the variable and constants of the entity which is the *iterated* entity.

$$\Big(\forall i \in \mathbb{N}_1^n\Big) \Big(\mathit{inputs}(\mathit{template}(Q))(i) = (j, TV) \in \mathit{arg}^{\mathrm{TV}}(TP) \land \mathit{iterator}(\mathit{template}(Q)) = i\Big) \tag{4.53a}$$
$$\implies \mathit{inputs}(Q)(i) \in \mathit{variables}(\mathit{iterated}(P))$$

$$\Big(\forall i \in \mathbb{N}_1^n\Big) \Big(\mathit{inputs}(\mathit{template}(Q))(i) = (j, TC) \in \mathit{arg}^{\mathrm{TC}}(TP) \land \mathit{iterator}(\mathit{template}(Q)) = i\Big) \tag{4.53b}$$
$$\implies \mathit{inputs}(Q)(i) \in \mathit{constants}(\mathit{iterated}(P))$$

Nested Processes

Nested processes are specified with the *processes* mapping. Similarly to equations, one nested process in the process template corresponds to a set of nested processes in the process instance. In the process-based formalism, nested processes are specified as a single list and not as a list of sets, because the system can infer the the placement of nested processes by the order in which they are specified.

The number of nested processes in the process template must correspond to the number of nested process sets in the process instance:

$$|\mathit{processes}(p)| = |\mathit{processes}(\mathit{template}(p))| \tag{4.54}$$

The type of the process specified as nested process in a process instance must be compatible with the template of the nested process specified in the process template:

$$\Big(\forall i \in \mathbb{N}_1^{|\mathit{processes}(P)|}\Big) \Big(\forall \hat{P} \in \mathit{processes}(P)_i\Big) \mathit{ancestor}^{\mathrm{P}} \Big(\mathit{template}(\hat{P}), \mathit{processes}(\mathit{template}(P))_i\Big) \tag{4.55}$$

The reason why nested processes are grouped into sets is to accommodate for the use of iterators. The nested process which are not iterated contain a single process instance, whereas the nested processes which are iterated contain as many process instances as there are entity instances in the argument for which they are iterated:

$$\Big(\forall i \in \mathbb{N}_1^{|\mathit{processes}(p)|}\Big) \mathit{processes}^{\mathrm{I}}(\mathit{template}(p))(i) = 0 \implies |\mathit{processes}(p)_i| = 1 \tag{4.56a}$$

$$\Big(\forall i \in \mathbb{N}_1^{|\mathit{processes}(p)|}\Big) \tag{4.56b}$$
$$\mathit{processes}^{\mathrm{I}}(\mathit{template}(p))(i) = j > 0 \implies |\mathit{processes}(p)_i| = \big|\mathit{arguments}(P)_j\big|$$

Each nested process specified in a process template specifies the mapping between the arguments of the containing process and the nested process. Once a process is instantiated with specific arguments, these constraints translate to exact requirements of the entity instances that must appear as arguments to processes that are specified as nested processes of other processes.

The following statements formally state these requirements for both non-iterated and iterated nested processes:

$$\begin{aligned}&\left(\forall i \in \mathbb{N}_1^{|processes(P)|}\right) processes^{\mathrm{I}}(template(P))(i) = 0 \implies \\ &\quad \left(\exists! \hat{P} \in processes(P)_i\right) \left(\forall j \in \mathbb{N}_1^{\left|arguments(\hat{P})\right|}\right) \\ &\quad arguments(\hat{P})_j = arguments(P)_{processes^{\mathrm{A}}(template(P))(i,j)}\end{aligned} \tag{4.57a}$$

$$\begin{aligned}&\left(\forall i \in \mathbb{N}_1^{|processes(P)|}\right) processes^{\mathrm{I}}(template(P))(i) = k > 0 \implies \\ &\quad \left(\forall j \in \mathbb{N}_1^{\left|arguments(\hat{P})\right|}\right) processes^{\mathrm{A}}(i,j) \neq k \implies \\ &\quad \left(\exists! \hat{P} \in processes(P)_i\right) arguments(\hat{P})_j = arguments(P)_{processes^{\mathrm{A}}(i,j)}\end{aligned} \tag{4.57b}$$

$$\begin{aligned}&\left(\forall i \in \mathbb{N}_1^{|processes(P)|}\right) processes^{\mathrm{I}}(template(P))(i) = k > 0 \implies \\ &\quad \left(\forall j \in \mathbb{N}_1^{\left|arguments(\hat{P})\right|}\right) processes^{\mathrm{A}}(i,j) = k \implies \\ &\quad \left(\forall E \in arguments(P)_{processes^{\mathrm{A}}(i,j)}\right) \left(\exists \hat{P} \in processes(P)_i\right) arguments(\hat{P})_j = \{E\}\end{aligned} \tag{4.57c}$$

Nested processes are typically interpreted as components of the process in which they are nested. From this interpretation of nested processes arises a need to disallow cyclic nesting of processes. This requirement is satisfied by default by all models, because it is a consequence of the requirements of the library.

As enforced by Property (4.28) in Section 4.2.4, the process templates can not be nested cyclically. Since process instances obey the specifications in the process template, this requirement directly transfers to process instances.

5 From Qualitative to Quantitative Models

In the previous chapter, we introduced process-based models especially their qualitative aspect comprising compartments, entities and processes. At a lower, quantitative level, these model elements include variables, constants and equations. Here, we focus on the task of bringing together all quantitative properties from different model elements. Put together, they represent the quantitative representation of the process-based model.

The content of a process-based model is taxonomically structured with compartments. Entities and processes are distributed among compartments for better organization and more realistic representation. To bring the quantitative model properties together, we firs define mappings between models and their components in terms of compartments, entities, and processes. To this end, we define a mapping $\mathit{compartments}^{\mathrm{R}} : \boldsymbol{\mathcal{M}} \rightarrow \mathscr{P}(\boldsymbol{\mathcal{K}})$ which to each model adjoins the set of all compartments that lie beneath it in the compartmental taxonomy. We define *compartments*$^{\mathrm{R}}$ as the transitive closure of the *compartments* mapping.

$$\mathit{compartments}^{\mathrm{R}} = \mathit{compartments}^{*} \tag{5.1}$$

Figure 5.1(a) shows a model containment hierarchy including the model (M), compartments (K_i), entities (E_j) and processes (P_k). The set of all nested compartments of a given model is highlighted in Figure 5.1(b).

Each model can contain entities and processes that are contained at the model level and entities and processes which are embedded within nested compartments. The entities and processes contained at the model level are represented with the *entities*$(\cdot)$ and *processes*$(\cdot)$ mappings. We define the mappings *entities*$^{\mathrm{R}}(\cdot)$ and *processes*$^{\mathrm{R}}(\cdot)$ to collect all entities and processes which are contained in a model.

$$\mathit{entities}^{\mathrm{R}} : \boldsymbol{\mathcal{M}} \rightarrow \mathscr{P}(\boldsymbol{\mathcal{E}}) \tag{5.2a}$$
$$\mathit{processes}^{\mathrm{R}} : \boldsymbol{\mathcal{M}} \rightarrow \mathscr{P}(\boldsymbol{\mathcal{P}}) \tag{5.2b}$$

The *entities*$^{\mathrm{R}}(\cdot)$ and *processes*$^{\mathrm{R}}(\cdot)$ functions are defined using the *compartments*$^{\mathrm{R}}(\cdot)$ mapping:

$$\mathit{entities}^{\mathrm{R}}(M) = \left\{ E \in \boldsymbol{\mathcal{E}} \,\middle|\, E \in \mathit{entities}(M) \vee (E \in \mathit{entities}(K) \wedge K \in \mathit{compartments}^{\mathrm{R}}(M)) \right\} \tag{5.3a}$$

$$\begin{aligned}\mathit{processes}^{\mathrm{R}}(M) = \Big\{ P \in \boldsymbol{\mathcal{P}} \,\Big|\, & P \in \mathit{processes}(M) \vee (P \\ & \in \mathit{processes}(K) \wedge K \in \mathit{compartments}^{\mathrm{R}}(M)) \Big\}\end{aligned} \tag{5.3b}$$

The sets of all nested entities and processes of a given model are highlighted in Figure 5.1(c) and Figure 5.1(d) respectively. Furthermore, we define functions that map a model to the sets of all variables, constants, and equations contained in the model.

$$\mathit{variables}^{\mathrm{R}} : \boldsymbol{\mathcal{M}} \rightarrow \mathscr{P}(\boldsymbol{\mathcal{V}}) \tag{5.4a}$$
$$\mathit{constants}^{\mathrm{R}} : \boldsymbol{\mathcal{M}} \rightarrow \mathscr{P}(\boldsymbol{\mathcal{C}}) \tag{5.4b}$$
$$\mathit{equations}^{\mathrm{R}} : \boldsymbol{\mathcal{M}} \rightarrow \mathscr{P}(\boldsymbol{\mathcal{Q}}) \tag{5.4c}$$

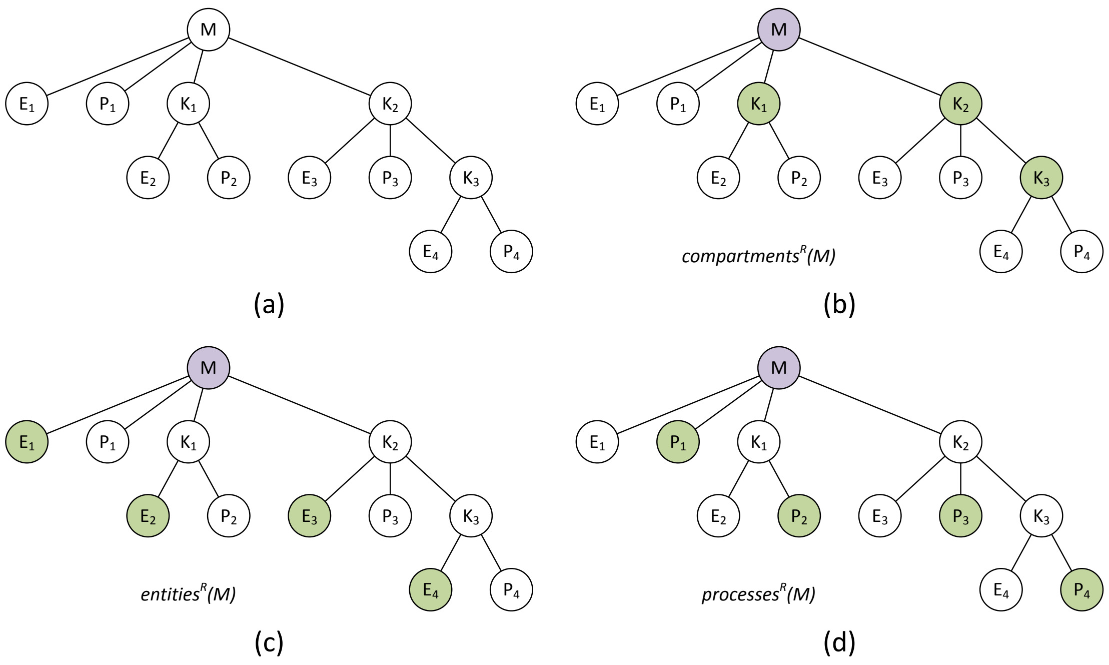


Figure 5.1: A model containment hierarchy highlighting nested (b) compartments, (c) entities and (d) processes.

The $\mathit{variables}^{\mathrm{R}}(\cdot)$, $\mathit{constants}^{\mathrm{R}}(\cdot)$ and $\mathit{equations}^{\mathrm{R}}(\cdot)$ functions are defined using the $\mathit{entities}^{\mathrm{R}}(\cdot)$ and $\mathit{processes}^{\mathrm{R}}(\cdot)$ mappings:

$$\mathit{variables}^{\mathrm{R}}(M) = \left\{ V \in \boldsymbol{\mathcal{V}} \,\middle|\, V \in \mathit{variables}(E) \wedge E \in \mathit{entities}^{\mathrm{R}}(M) \right\} \tag{5.5a}$$

$$\mathit{constants}^{\mathrm{R}}(M) = \left\{ C \in \boldsymbol{\mathcal{C}} \,\middle|\, C \in \mathit{constants}(S) \wedge S \in (\mathit{entities}^{\mathrm{R}}(M) \cup \mathit{processes}^{\mathrm{R}}(M)) \right\} \tag{5.5b}$$

$$\mathit{equations}^{\mathrm{R}}(M) = \left\{ Q \in \boldsymbol{\mathcal{Q}} \,\middle|\, Q \in \mathit{equations}(P) \wedge P \in \mathit{processes}^{\mathrm{R}}(M) \right\} \tag{5.5c}$$

The sets of all variables, constants, and equations of an example model are highlighted in Figure 5.2(b), Figure 5.2(c), and Figure 5.2(d) respectively.

In section 4.3, we introduced a categorization of model variables based on the role of the variable, which we illustrated in Figure 4.1. With this categorization, we partitioned variables into *exogenous* and *endogenous*, and then further partitioned *endogenous* variables into *auxiliary* and *state* variables. Different variable types serve a different purpose in the model. We define the following mappings that gather all variables that appear in a model by type:

$$\mathit{exogenous}(M) = \left\{ V \in \mathit{variables}^{\mathrm{R}}(M) \,\middle|\, \mathit{role}(V) = \text{exogenous} \right\} \tag{5.6a}$$

$$\mathit{endogenous}(M) = \left\{ V \in \mathit{variables}^{\mathrm{R}}(M) \,\middle|\, \mathit{role}(V) = \text{endogenous} \right\} \tag{5.6b}$$

$$\mathit{auxiliary}(M) = \left\{ V \in \mathit{variables}^{\mathrm{R}}(M) \,\middle|\, \mathit{auxiliary}(V) = \text{true} \right\} \tag{5.6c}$$

$$\mathit{state}(M) = \left\{ V \in \mathit{variables}^{\mathrm{R}}(M) \,\middle|\, \mathit{state}(V) = \text{true} \right\} \tag{5.6d}$$

Process instances within a model can appear either as top-level processes or nested processes. A top-level process is not included as a nested process in any other process in the model, whereas nested process only appear as subcomponents of other more complex processes. We express the set of all top-level processes $\mathit{processes}^{\mathrm{TOP}}$ of a model M formally:

$$\mathit{processes}^{\mathrm{TOP}}(M) = \left\{ P \in \mathit{processes}^{R}(M) \,\middle|\, \neg \left(\exists \overline{P} \in \mathit{processes}^{R}(M)\right) P \in \mathit{processes}(\overline{P}) \right\} \tag{5.7}$$

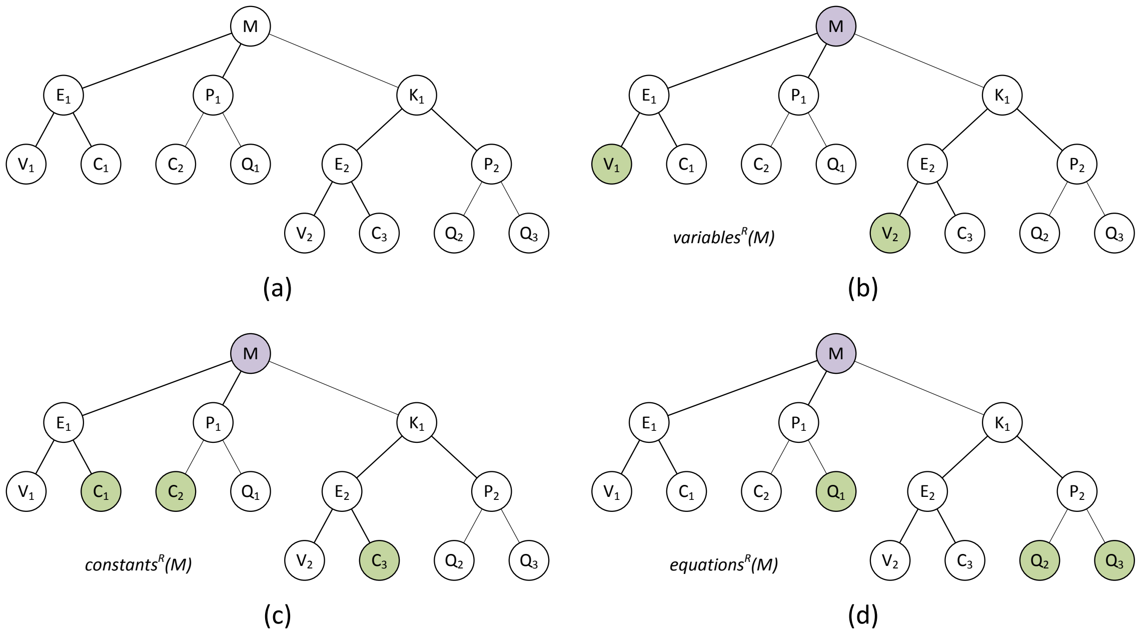


Figure 5.2: A hypothetical model containment hierarchy highlighting the variables, constants, and equations in the model.

5.1 Model Structure, Parameters, and Completeness

Here, we define two important constituents of a process-based model, *model structure* and *model parameters*. The later are numerical parameters that comprise initial values of the model state variables and values of the model constants. In other words, the parameters of a model are defined with the restriction of the function $\mathit{initial}$ to the set $\mathit{state}(M)$ (denoted as $\mathit{initial}|_{\mathit{state}(M)}$) and the restriction of the function value to the set $\mathit{constants}(M)$ (denoted as $\mathit{value}|_{\mathit{constants}(M)}$). Given these two restrictions, we define the model parameters as a union of the state variables and constants of the model:

$$\mathit{parameters}(M) = \mathit{state}(M) \cup \mathit{constants}(M), \quad M \in \boldsymbol{\mathcal{M}} \tag{5.8}$$

All the other information provided in the definition of a process-based model is referred to as model structure.

The example process-based models from the previous chapter have completely specified both the structure and parameters. In this section, we will consider incompletely specified process-based models, where parts of the parameters and structure specification are missing.

5.2 Incomplete Model Parameters

The model parameters are completely specified (or complete) if the functions that specify model parameters ($\mathit{initial}$ and values) are defined on the whole sets of parameters ($\mathit{state}(M)$ and $\mathit{constants}(M)$). If some of the model parameters are left undefined, then the functions that specify model parameters are missing some mappings. In continuation, we define subsets of $\mathit{state}(M)$ and $\mathit{constants}(M)$ that correspond to the known and to the unknown initial and constant parameters.

The set $\mathit{state}^{\oplus}(M)$ is the set of state variables whose initial values are *specified*. The set $\mathit{state}^{\ominus}$ is the set of state variables for which the initial values are *unspecified*. Together, $\mathit{state}^{\oplus}(M)$ and $\mathit{state}^{\ominus}$ form a partition of $\mathit{state}(M)$, i.e., $\mathit{state}(M) = \mathit{state}^{\oplus}(M) \cup \mathit{state}^{\ominus}(M)$ and $\mathit{state}^{\oplus}(M) \cap \mathit{state}^{\ominus}(M) = \varnothing$.

Example 5.1: A simplified model of a lake ecosystem defined by using the hierarchical version of the aquatic ecosystem library.

```
model completeLakeModel : AquaticHierarchicalLibrary;
entity phyto : PrimaryProducer {
    vars:
        conc{role: endogenous; initial: 10},
        nutrientLim;
    consts:
        maxGrowthRate = 0.5,
}
entity phosphorus : Nutrient {
    vars:
        conc{role: endogenous; initial: 3};
    consts:
        halfSaturation = 0.02,
        alpha = 0.1;
}
entity nitrogen : Nutrient {
    vars:
        conc;
    consts:
        halfSaturation = 0.2,
        alpha = 0.7;
}
process growth(phyto, [phosphorus,nitrogen]) : Growth {
    processes:
        phosphorusLim, nitrogenLim;
}
process phosphorusLim(phyto, phosphorus) : MonodLimitation{}
process nitrogenLim(phyto, nitrogen) : Monod2Limitation{}
```

$$initial(V) = \begin{cases} r \in \mathbb{R} & V \in state^{\oplus}(M) \\ \text{undefined} & V \in state^{\ominus}(M) \end{cases} \tag{5.9}$$

The set $constants^{\oplus}(M)$ is the set of constants with *specified* values. The set $constants^{\ominus}$ is the set of constants with *unspecified* values. Together, $constants^{\oplus}(M)$ and $constants^{\ominus}$ form a partition of $constants(M)$, i.e., $constants(M) = constants^{\oplus}(M) \cup constants^{\ominus}(M)$ and $constants^{\oplus}(M) \cap constants^{\ominus}(M) = \varnothing$.

$$value(C) = \begin{cases} r \in \mathbb{R} & C \in constants^{\oplus}(M) \\ \text{undefined} & C \in constants^{\ominus}(M) \end{cases} \tag{5.10}$$

The sets of all model parameters with specified values $parameters^{\oplus}$ and all unspecified model parameters $parameters^{\ominus}$ are defined as:

$$parameters^{\oplus}(M) = state^{\oplus}(M) \cup constants^{\oplus}(M) \tag{5.11a}$$
$$parameters^{\ominus}(M) = state^{\ominus}(M) \cup constants^{\ominus}(M) \tag{5.11b}$$

Within the process-based formalism, we declare that the value of a numerical parameter in a given model is unspecified, by assigning it the special value *null* instead of the usual numerical value. The *phyto* entity from Example 5.1, contains a state variable *conc* with

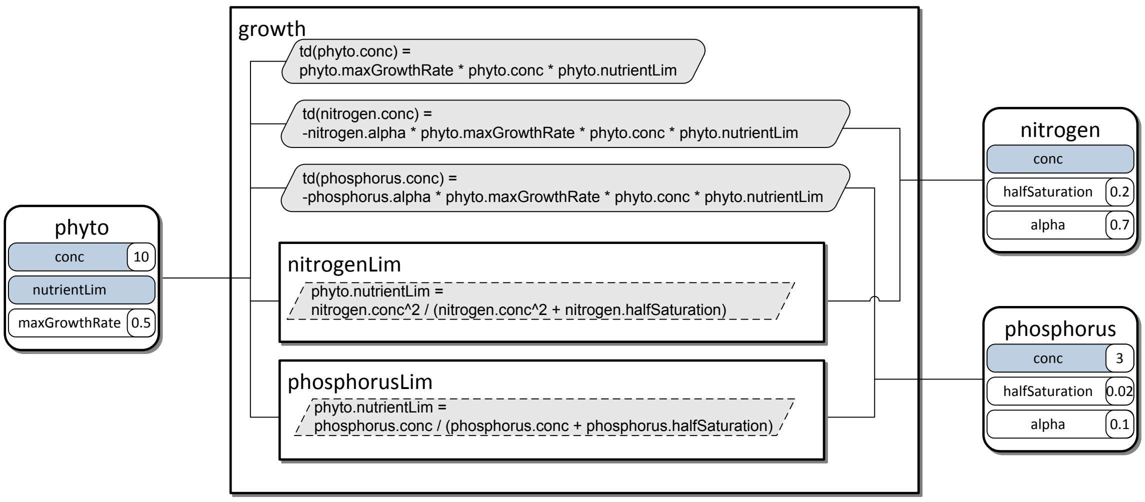

growth
td(phyto.conc) =
phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
td(nitrogen.conc) =
-nitrogen.alpha * phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
td(phosphorus.conc) =
-phosphorus.alpha * phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
nitrogenLim
phyto.nutrientLim =
nitrogen.conc^2 / (nitrogen.conc^2 + nitrogen.halfSaturation)
phosphorusLim
phyto.nutrientLim =
phosphorus.conc / (phosphorus.conc + phosphorus.halfSaturation)
phyto
conc 10
nutrientLim
maxGrowthRate 0.5
nitrogen
conc
halfSaturation 0.2
alpha 0.7
phosphorus
conc 3
halfSaturation 0.02
alpha 0.1

Figure 5.3: A complete model.

an initial value of 10 and a constant *maxGrowthRate* with a value of 0.5. The complete model from Example 5.1 is illustrated in Figure 5.3. If the values of these two numerical parameters were not specified, the definition of *phyto* would become:

entity *phyto* : *PrimaryProducer* {
  vars:
    conc {role: *endogenous*; initial: null},
    nutrientLim;
  consts:
    maxGrowthRate = null;
}

Instead of a value function specifying their values, parameters with unspecified values have a function *fitting range* defining the bounds on the expected parameter value. The fitting ranges of the unspecified parameter values are defined using the $range^{\mathrm{F}}$ function.

$$range^{\mathrm{F}}(V) \mapsto [a,b], \quad a,b \in \mathbb{R},\ V \in state^{\ominus}(M) \tag{5.12a}$$
$$range^{\mathrm{F}}(C) \mapsto [a,b], \quad a,b \in \mathbb{R},\ C \in constants^{\ominus}(M) \tag{5.12b}$$

In the process-based formalism, we specify the fitting ranges using the *fit_range* property. When specifying the fitting range for initial values of state variables, we simply add the additional property *fit_range* which has a value of type interval to the state variable. When specifying the fitting range for the value of a model constant, we extend the definition of the constant with a block specification enclosed in curly brackets. Within the block specification (which is the usual way of specifying properties of variables for example), we include the *fit_range* property specified as an interval.

The suitable fitting range for the *conc* state variable from Example 5.1 is, for example, the interval $[0.60, 0.61]$, whereas the fitting range for the *maxGrowthRate* is $[0.05, 3]$. The definition of *phyto* with this information included is as follows:

entity *phyto* : *PrimaryProducer* {
  vars:
    conc {role: *endogenous*; initial: null; *fit_range*:<0.60, 0.61>},
    nutrientLim;
  consts:
    maxGrowthRate {*fit_range*:<0.05,3>} = null;
}

The fitting ranges are specified by the modeling expert and usually come from modeling knowledge in the specific domain. The fitting ranges for the examples used throughout the dissertation are taken from Atanasova et al. (2006c).

Note that ranges for the values of variables and constants can be specified in libraries. If a model is defined using a library, we can make use of these ranges. The fitting range can then be defined to match the range specified in the library.

5.3 Incomplete Model Structure

Similarly to the completeness of the model parameters, the model structure is complete when all of the corresponding mappings are well and completely defined. If some of the mappings are incomplete, we refer to the corresponding model structure as incomplete. We distinguish here two types of incomplete model structure: incomplete process arguments and incomplete process types.

5.3.1 Incomplete Process Arguments

The arguments of a process in a process-based model are specified with the mapping *arguments*. This mapping associates each process instance to a tuple of sets of entity instances: $P \mapsto (E_1, E_2, \ldots, E_n)$, where $E_i \subset \boldsymbol{\mathcal{E}}$. Note that we can also consider them as a function that maps integers to sets of entities $i \mapsto E_i$, where the integer i is the index of the entity set E_i in the arguments tuple:

$$arguments(P) = \begin{pmatrix} 1 & 2 & \cdots & n \\ E_1 & E_2 & \cdots & E_n \end{pmatrix}$$

Following this definition, we can now represent process *arguments* as:

$$arguments : \boldsymbol{\mathcal{P}} \to (\mathbb{N}_1^k \to \mathscr{P}(\boldsymbol{\mathcal{E}})) \tag{5.13}$$

We uniquely identify an argument position with the pair (P,i), where P is the process instance to which the argument belongs and i is the position of the argument within the argument list of the process P. We then form the set of all argument positions in a given model M (denoted $positions(M)$) as:

$$positions(M) = \left\{ (P,i) \middle| P \in processes^{\mathrm{R}}(M) \wedge i \in \mathbb{N}_1^{|arguments(P)|} \right\} \tag{5.14}$$

Using this definition of $positions(M)$, we can define all the arguments in a given model as a mapping $arguments(M) : positions(M) \to \mathscr{P}(E)$ defined with $(P,i) \mapsto arguments(P)(i)$.

We can then summarize $arguments(M)$ as:

$$arguments(M) = \left\{ (P,i) \mapsto E \middle| P \in processes^{\mathrm{R}}(M) \wedge i \in \mathbb{N}_1^{|arguments(P)|} \wedge E = arguments(P)(i) \right\} \tag{5.15}$$

In an incomplete process-based model, we allow some of the bindings of the argument mappings to be left unspecified. With the missing argument mapping, we indicate that we are not able to specify which entities participate in the corresponding process.

We define subsets of the index set $positions(M)$ which correspond to the specified and unspecified argument bindings.

The set $positions^{\oplus}(M)$ is the set of argument position for which the argument binding is *specified.* The set $positions^{\ominus}$ is the set of argument positions for which the argument binding is *unspecified.* Together, $positions^{\oplus}(M)$ and $positions^{\ominus}(M)$ form a partition of $positions(M)$, i.e., $positions(M) = positions^{\oplus}(M) \cup positions^{\ominus}(M)$ and $positions^{\oplus}(M) \cap positions^{\ominus}(M) = \varnothing$.

$$arguments(pos) = \begin{cases} E \subset \boldsymbol{\mathcal{E}} & pos \in positions^{\oplus}(M) \\ \text{undefined} & pos \in positions^{\ominus}(M) \end{cases} \tag{5.16}$$

Note however, that in many cases, we are able to provide partial information about a missing binding. The process-based formalism allows us to specify a set of entity instances that we find suitable to be used for the particular argument. This set of entities forms the *upper bound* of the argument. For arguments which have cardinality one, the upper bound is the only reasonable constraint. However, for arguments which have cardinality larger than one, it also makes sense to specify a *lower bound.* The lower bound defines the set of entities that are mandatory for that argument. This is useful in the case where we are certain that particular entities are involved in the process, but we are not certain whether they are the only entities that play a role in the process.

We define two functions: lower bound $bound^{\mathrm{L}}$ and upper bound $bound^{\mathrm{U}}$, which for each process argument position that has an unspecified argument, provide the constraints on the allowed values in the form of lower and upper bound.

$$bound^{\mathrm{L}} : positions(M) \rightarrow \mathscr{P}(\boldsymbol{\mathcal{E}}) \tag{5.17a}$$
$$bound^{\mathrm{U}} : positions(M) \rightarrow \mathscr{P}(\boldsymbol{\mathcal{E}}) \tag{5.17b}$$

Within the process-based formalism, we declare that a process argument has an unspecified binding by providing the lower and upper bounds instead of the value of the arguments. For example, the *growth* process from Example 5.1 was defined as a complete process:

process *growth*(*phyto*, [*phosphorus*, *nitrogen*]) : *Growth* {
  // body omitted . . .
}

If the second argument, which denotes the nutrients were unspecified, but *phosphorus* and *nitrogen* were candidates for participating in the *growth* process, then its definition would be:

process *growth*(*phyto*, [[], [*phosphorus*, *nitrogen*]]) : *Growth* {
  // body omitted . . .
}

Unspecified argument bindings are denoted within a pair of square brackets, and consist of two parts: the first part is the lower bound surrounded by square brackets and the second part is the upper bound surrounded by square brackets. In this example, the lower bound is an empty set (denoted by []), whereas the upper bound is the two-element set of *phoshprus* and *nitrogen* (denoted by [phosphorus, nitrogen]).

We can leave an argument completely unspecified by giving it the value [[],[all]]. If this value were assigned to the nutrients argument in the previous example, then this argument would become completely unspecified. What is known is that the argument in that position has to be of type *Nutrient* and knowing the instance entities of type *Nutrient* in the model, defines the set of possible choices for that argument. In the model in Example 5.1, we have two instance entities of type *Nutrient*: *nitrogen* and *phosphorus*, therefore the possible choices for a suitable argument would be made from [nitrogen, phosphorus].

5.3.2 Incomplete Process Types

Recall from the previous chapter, that the entity and process templates form inheritance taxonomies. Following them the templates can be classified at different level of abstractions. The templates in the inner taxonomy nodes are *abstract* templates representing general concepts for grouping and organizing knowledge. The templates at the leaf taxonomy nodes are *concrete* templates and serve to create instances in a process-based model.

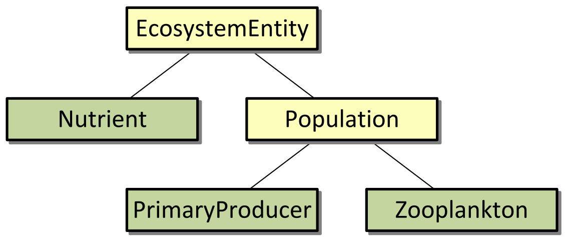


Figure 5.4: Taxonomy of entity templates depicting abstract entity templates (yellow) and concrete entity templates (green).

All entity instances that appear in a process-based model are created from concrete entity templates. Even though abstract entity templates are not directly used to create instances, they have an important role. By using abstract entity templates as types for process arguments, we creating more general processes that can handle different concrete types of arguments.

Figure 5.4 presents an example of an entity taxonomy. This is an extension to the aquatic ecosystems library presented in Example 3.9 and depicted in Figure 3.3(a), which includes the additional entity template *Zooplankton*. *Phytoplankton* and *Zooplankton* are both types of *Population*.

We define two predicates ***abstract*** and ***concrete***, which answer the question whether a given entity template $E \in \boldsymbol{\mathcal{E}}$ is abstract or concrete.

$$abstract(E) = \text{true} \iff \left(\exists \widehat{E} \in \boldsymbol{\mathcal{E}}\right) super^{\mathrm{E}}(E, \widehat{E}) \tag{5.18a}$$

$$concrete(E) = \text{true} \iff \neg\left(\exists \widehat{E} \in \boldsymbol{\mathcal{E}}\right) super^{\mathrm{E}}(E, \widehat{E}) \tag{5.18b}$$

For each abstract entity template, we define its set of *concretizations*. A concretization of an abstract entity template is a concrete entity template that lies beneath the abstract entity template in the taxonomy of entity templates. Formally:

$$concretizations(E) = \left\{\widehat{E} \in \boldsymbol{\mathcal{E}} \middle| ancestor^{\mathrm{E}}(E, \widehat{E}) \wedge concrete(\widehat{E})\right\} \tag{5.19}$$

The *EcosystemEntity* from Figure 5.4, for example, has a set of concretizations consisting of three elements: *Nutrient*, *PrimaryProducer* and *Zooplankton*.

Process templates are also organized in a taxonomy. The taxonomy of process templates is an alternatives taxonomy. The inner nodes in the taxonomy represent *conceptual* processes, where each conceptual process is a generalization for a class of processes. An example of a conceptual process is the *Temperature Growth Influence* process given in Figure 5.5. This process is a conceptualization of the influence that temperature has on the growth of phytoplankton. It specifies the arguments that are involved in such a process, but it does not specify any equations. Equations are the mathematical representation of the process, and temperature can influence the growth of phytoplankton in many different ways, which will be reflected in different equations. The different mathematical specifications of the influence that a process has is specified through *specific* process templates.

Specific process templates are the leaf nodes in the taxonomy of process templates. Temperature influence can be represented as *No temperature limitation* which is the trivial case and *Temperature limitation* which indicates that the influence that temperature exerts on the growth of phytoplankton is some form of limitation. The temperature limitation is itself a conceptual process because the actual mathematical form of the limitation can be specified with different limitation functions. This is the purpose of specific processes.

A specific process performs the role that is conceptualized by its parent process template by providing one mathematical form. Each mathematical form is one alternative

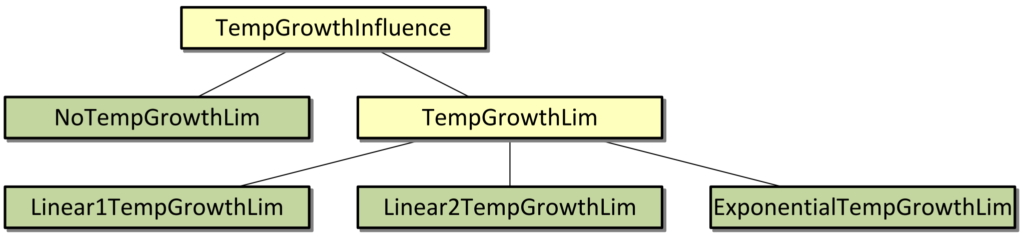


Figure 5.5: Taxonomy of process templates depicting conceptual process templates (yellow) and specific process templates (green).

for the role of the conceptual process. In Figure 5.5, the conceptual process *Temperature Growth Limitation* has three mathematical alternatives. The first alternative denoted as *Linear 1* is given with the formula $tempGrowthLim = temperature/refTemp$, with the reference temperature $refTemp$ as parameter. The second alternative is denoted as *Linear 2* and is also a linear dependency given with the formula $tempGrowthLim = (temperature - minTemp)/(refTemp - minTemp)$, where the distance to the known minimal temperature $minTemp$ is the driving force. The third alternative is denoted as *Exponential* and given with the formula $tempGrowthLim = pow(theta, temperature - refTemp)$.

We define two predicates *conceptual* and *specific*, which answer the question whether a given process template $P \in \mathcal{P}$ is conceptual or specific.

$$conceptual(P) = \text{true} \iff \left(\exists \widehat{P} \in \mathcal{P}\right) super^{\mathrm{P}}(P, \widehat{P}) \tag{5.20a}$$

$$specific(P) = \text{true} \iff \neg \left(\exists \widehat{P} \in \mathcal{P}\right) super^{\mathrm{P}}(P, \widehat{P}) \tag{5.20b}$$

For each conceptual process template, we define its set of *specifications*. A specification of a conceptual process template is a specific process template that lies beneath the conceptual process template in the taxonomy of process templates. Formally:

$$specifications(P) = \left\{ \widehat{P} \in \mathcal{P} \middle| ancestor^{\mathrm{P}}(P, \widehat{P}) \wedge specific(\widehat{P}) \right\} \tag{5.21}$$

The *TempGrowthIncluence* process from Figure 5.5, for example, has a set of specifications consisting of four elements: *NoTempGrowthLim*, *Linear1TempGrowthLim*, *Linear2TempGrowthLim*, and *ExponentialTempGrowthLim*.

All complete models that were discussed so far contained only process instances that had as templates, specific process templates. In these models, each process instance is committed to a particular mathematical form. Process instances can have as templates conceptual process templates. In that case, the process instance is *incomplete* in the sense that it is lacking the necessary equations to be a fully quantified process. We denote this type of incompleteness as incomplete process type.

Unlike incomplete parameter values and incomplete process arguments, for which the incompleteness is expressed as a missing definition, for which we substitute a constraint designating the allowed values, here, the *template* function is defined. The definition, however, is not considered a complete definition. In order for the *template* function to be considered complete, it is required that it maps to specific process template.

In Chapter 3, we defined the function $processes^{\mathrm{R}} : \mathcal{M} \rightarrow \mathscr{P}(\mathcal{P})$ which yields the set of all processes contained in a model. Furthermore, we partition the set of all processes in a model into processes with specific templates—$processes^{\oplus}$ and conceptual templates—$processes^{\ominus}$:

$$processes^{\oplus}(M) = \left\{ P \in processes^{\mathrm{R}}(M) \middle| specific(P) = \text{true} \right\} \tag{5.22a}$$

$$processes^{\ominus}(M) = \left\{ P \in processes^{\mathrm{R}}(M) \middle| conceptual(P) = \text{true} \right\} \tag{5.22b}$$

The set $processes^{\ominus}(M)$ is the set of processes with incomplete types. For each process P with an incomplete type, the set of specifications of that process $specifications(P)$ is the set of alternative forms that the process can acquire.

In the process-based formalism, a process with incomplete process type is simply defined by providing a process template that is a conceptual process template. For example, the definition of the limitation of phytoplankton growth by phosphorus is defined in Example 5.1 is given as:

process *phosphorusLim*(*phyto*, *phosphorus*) : *MonodLimitation*{}

If the *phosphorusLim* process is defined as a conceptual process, then the definition would be as follows:

process *phosphorusLim*(*phyto*, *phosphorus*) : *NutrientLimitation*{}

5.3.3 Structurally Complete and Incomplete processes

A process with a completely specified type and arguments is a structurally complete process. In contrast, a process with incomplete structure can have incomplete type or arguments specification. Hence, a structurally incomplete process contains one or two types of incompleteness. Note however, that both structurally complete and incomplete processes may contain incompletely specified parameters, so a structurally complete process is not necessarily a completely defined process.

We define a *completion* relation between a structurally complete process P and a structurally incomplete process $\widehat{P}$: $completion(P,\widehat{P})$ which we read P *is a completion of* $\widehat{P}$. Let $\widehat{P}$ be an arbitrary structurally incomplete process. In order for a structurally complete process P to be a completion of $\widehat{P}$, it must have *suitable* process type and *suitable* arguments. The process type of P is suitable if it is a completion of the process type of $\widehat{P}$. If the process type of $\widehat{P}$ is specific, then P must have the same process type. In other words, the following statement must hold:

$$template(P) \in completion(\widehat{P}) \tag{5.23}$$

The notion of suitable arguments is slightly more complex. First, all specified arguments of $\widehat{P}$ have to appear in P also. In addition, all unspecified arguments of $\widehat{P}$ must be specified in P and must lie between the lower and upper bound of that argument. In other words, the following statements regarding specified and unspecified arguments must hold:

$$\left(k \in \mathbb{N}_1^{|arguments(P)|}\right) (\widehat{P},k) \in positions^{\oplus}(M) \implies arguments_k(P) = arguments_k(\widehat{P}) \tag{5.24a}$$

$$\left(k \in \mathbb{N}_1^{|arguments(P)|}\right) (\widehat{P},k) \in positions^{\ominus}(M) \implies bound^{\mathrm{L}}(\widehat{P},k) \subset arguments_k(P) \subset bound^{\mathrm{U}}(\widehat{P},k) \tag{5.24b}$$

5.4 Incomplete Model Specification

In the previous sections we described three kinds of incompleteness of process-based models. A model which contains incompleteness of any kind is denoted as *incomplete model*. An incomplete model consist of entity and process instances, some of which contain incompleteness of one or more kinds.

The entities contained in an incomplete model are required to be structurally complete; they can only be incomplete in terms of model parameters. In other words, the number and type of entities in the incomplete model is given and fixed. Moreover, the type of each entity is always a concrete entity template. Finally, incomplete models can either have unspecified initial values of state variables or unspecified values of constants.

The processes in the incomplete model, on the other hand, can contain all three kinds of incompleteness. They can have incomplete parameter values, incomplete process argument bindings or incomplete (conceptual) process types. In the remainder of this section, we are going to introduce each type of incompleteness and illustrate it on an example.

5.4.1 Models with Complete Structure and Incomplete Parameters

The simplest type of incomplete model is a model with a complete model structure and incomplete numerical parameters. The incomplete model in Example 5.2 is based on the complete model from Example 5.1, with the initial values of *phyto.conc* and *phosphorus.conc* and the values of *phosphorus.halfSaturation* and *nitrogen.halfSaturation* omitted.

Example 5.2: A model with a complete structure and incomplete parameters.

```
incomplete model lakeModelIncompleteParameters : AquaticHierarchicalLibrary;
entity phyto : PrimaryProducer {
    vars:
        conc{role: endogenous; initial: null},
        nutrientLim;
    consts:
        maxGrowthRate = 0.5,
}
entity phosphorus : Nutrient {
    vars:
        conc{role: endogenous; initial: null};
    consts:
        halfSaturation = null,
        alpha = 0.1;
}
entity nitrogen : Nutrient {
    vars:
        conc;
    consts:
        halfSaturation = null,
        alpha = 0.7;
}
process growth(phyto, [phosphorus,nitrogen]) : Growth {
    processes:
        phosphorusLim, nitrogenLim;
}
process phosphorusLim(phyto, phosphorus) : MonodLimitation{}
process nitrogenLim(phyto, nitrogen) : Monod2Limitation{}
```

This definition of an incomplete model starts with a declaration which includes the keyword pair **incomplete model** followed by the name of the incomplete model (in this example *lakeModelWithoutParameters*), a colon (:), and the name of the library which contains the domain knowledge (*AquaticHierarchicalLibrary*), and ending with a semicolon (;).

Figure 5.6 shows a graphical representation of the same incomplete model. The specified initial values of state variables and values of constants are inscribed within rounded squares adjacent to the name of the attribute. The unspecified values are represented as empty red rounded squares indicating that this information is missing in the model.

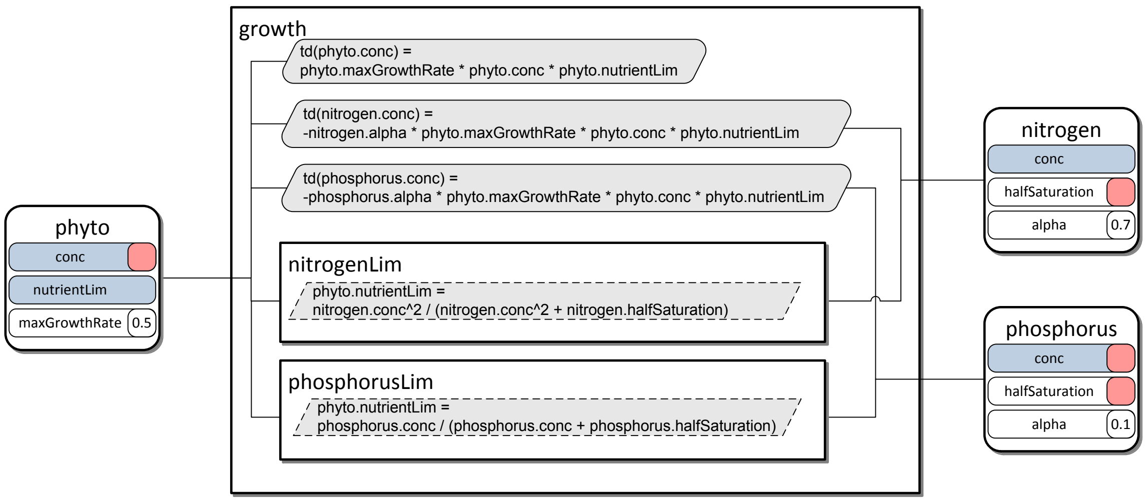

growth
td(phyto.conc) =
phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
td(nitrogen.conc) =
-nitrogen.alpha * phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
td(phosphorus.conc) =
-phosphorus.alpha * phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
nitrogenLim
phyto.nutrientLim =
nitrogen.conc^2 / (nitrogen.conc^2 + nitrogen.halfSaturation)
phosphorusLim
phyto.nutrientLim =
phosphorus.conc / (phosphorus.conc + phosphorus.halfSaturation)
phyto
conc
nutrientLim
maxGrowthRate 0.5
nitrogen
conc
halfSaturation
alpha 0.7
phosphorus
conc
halfSaturation
alpha 0.1

Figure 5.6: A model with incomplete numerical parameters.

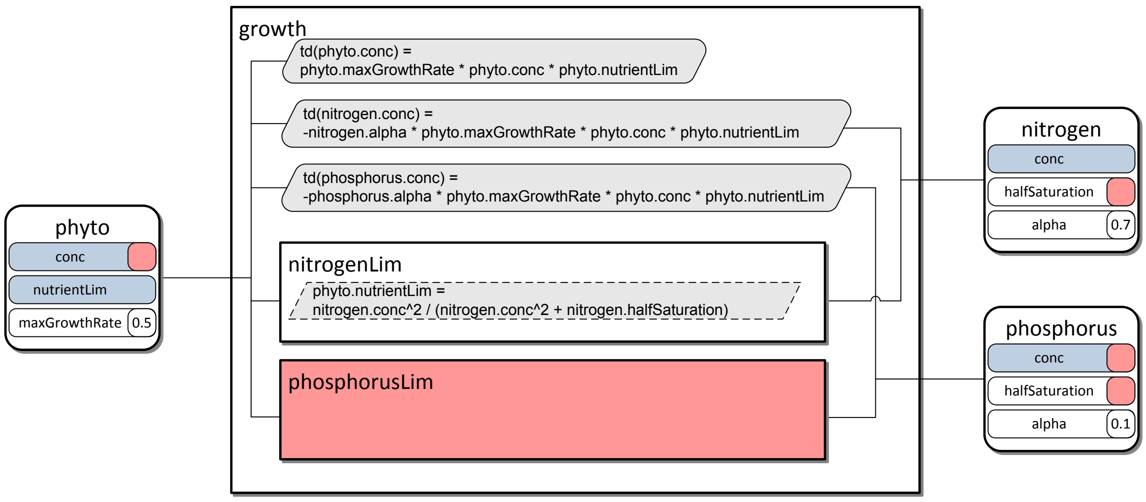

growth
td(phyto.conc) =
phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
td(nitrogen.conc) =
-nitrogen.alpha * phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
td(phosphorus.conc) =
-phosphorus.alpha * phyto.maxGrowthRate * phyto.conc * phyto.nutrientLim
nitrogenLim
phyto.nutrientLim =
nitrogen.conc^2 / (nitrogen.conc^2 + nitrogen.halfSaturation)
phosphorusLim
phyto
conc
nutrientLim
maxGrowthRate 0.5
nitrogen
conc
halfSaturation
alpha 0.7
phosphorus
conc
halfSaturation
alpha 0.1

Figure 5.7: A model with incomplete nested process types.

5.4.2 Models with Conceptual Nested Processes

Each process which is nested in another process is defined in the usual way, in the same compartment as the process which contains it (or at the model-level, if the containing process is defined at the model-level). Example 5.3 builds upon the incomplete model in Example 5.2 and illustrates incomplete process types with the *phosphorusLim* process.

The incomplete model is illustrated in Figure 5.7. Similarly to Figure 5.6, it contains unspecified parameters in the instance entities. In addition, the *nitrogenLim* process is only specified at a conceptual level (we refer to such specifications as conceptual processes) and therefore depicted as an empty red rectangle, drawing attention to its incomplete type.

In this scenario, the type and arguments of the surrounding process *growth* are specified, but the type of the nested process *phosphorusLim* is incomplete. In this situation, we can use a shorter way of declaring that the process type of the nested process is incomplete. The definition of the process *phosphorusLim* can be omitted altogether from the incomplete model, and the definition of the *growth* process then becomes:

process *growth*(*phyto*, [*phosphorus*,*nitrogen*]) : *Growth* {
    processes:
        NutrientLimitation, *nitrogenLim*;

Example 5.3: A model with incomplete nested process.

```
incomplete model lakeModelIncompleteParameters : AquaticHierarchicalLibrary;
entity phyto : PrimaryProducer {
    vars:
        conc{role: endogenous; initial: null},
        nutrientLim;
    consts:
        maxGrowthRate = 0.5,
}
entity phosphorus : Nutrient {
    vars:
        conc{role: endogenous; initial: null};
    consts:
        halfSaturation = null,
        alpha = 0.1;
}
entity nitrogen : Nutrient {
    vars:
        conc;
    consts:
        halfSaturation = null,
        alpha = 0.7;
}
process growth(phyto, [phosphorus,nitrogen]) : Growth {
    processes:
        phosphorusLim, nitrogenLim;
}
process phosphorusLim(phyto, phosphorus) : NutrientLimitation{}
process nitrogenLim(phyto, nitrogen) : Monod2Limitation{}
```

Example 5.4: A model with incomplete process arguments.

```
incomplete model lakeModelIncompleteArguments : AquaticHierarchicalLibrary;
entity phyto : PrimaryProducer {
    vars:
        conc{role: endogenous; initial: null},
        nutrientLim;
    consts:
        maxGrowthRate = 0.5,
}
entity phosphorus : Nutrient {
    vars:
        conc{role: endogenous; initial: null};
    consts:
        halfSaturation = null,
        alpha = 0.1;
}
entity nitrogen : Nutrient {
    vars:
        conc;
    consts:
        halfSaturation = null,
        alpha = 0.7;
}
process growth(phyto, [[],[phosphorus,nitrogen]]) : Growth {
    processes:
        NutrientLimitation;
}
```

}

Recall Property (4.57) from Section 4.3 according to which the arguments of a nested process are uniquely determined by the arguments of the containing process and the argument mapping. Therefore, these two alternative declarations of nested process incompleteness are equivalent. The only difference between them is that in this case, the nested process has an explicit name.

5.4.3 The Interplay of Incomplete Process Arguments and Nested Processes

If a top-level process has an argument that is not bound, and this argument is used as an iterator for a nested process, then it is not possible to know how many nested processes should be present. Consequently, it is not possible to specify them as a list of nested processes. The only reasonable specification in this case consists of providing the process template for the nested processes.

This scenario is presented in Example 5.4. Here, the *growth* process has an unspecified second argument and the definition of the *Growth* template in the *AquaticHierarchicalLibrary* mandates one nested process of type *NutrientLimitation* for each entity instance in the second argument:

```
template process Growth(pp : PrimaryProducer, ns : Nutrient<1, inf>) {
    processes:
        NutrientLimitation(pp, <n:ns>);
```

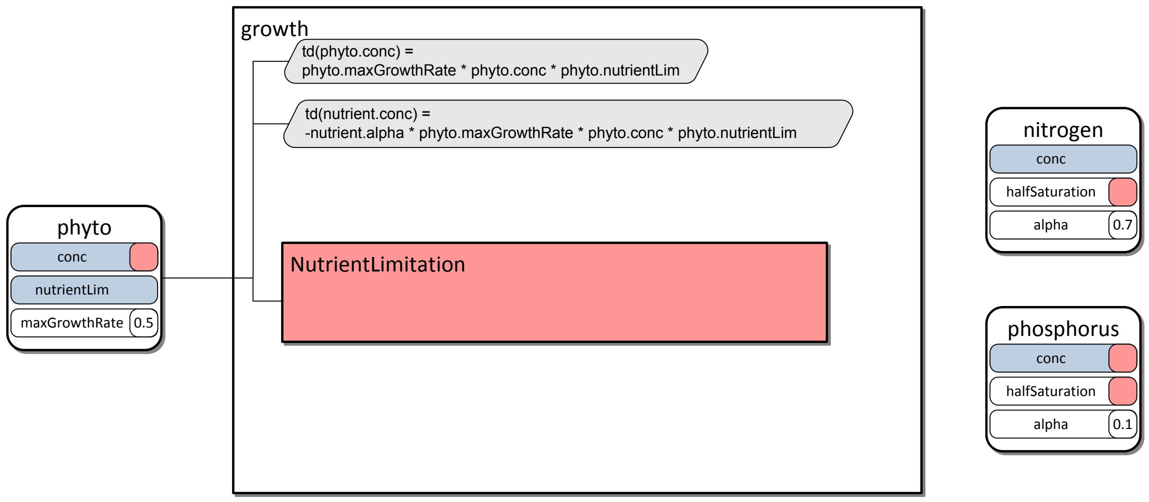


Figure 5.8: A model with incomplete process arguments.

```
    equations:
        td(pp.conc) = pp.maxGrowthRate ∗ pp.nutrientLim ∗ pp.conc,
        td(<n:ns>.conc) = n.alpha ∗ pp.maxGrowthRate ∗ pp.nutrientLim ∗ pp.conc;
}
```

We visualize this incomplete model as presented in Figure 5.8. The inclusion of *phyto* in the *growth* process is specified as indicated by the lines which connect *phyto* to the different parts of the *growth* process. *Nitrogen* and *phosphorus* are specified only as candidates for inclusion in the *growth* process and their participation is not certain. Therefore, they are not visually connected to the *growth* process and its constituents. The nested processes of the *growth* process are specified only by their type *NutrientLimitation* and their number and specific form is not determined.

5.4.4 Models with Incomplete Top-Level Processes

In many modeling scenarios, some of the top-level processes are specified at a conceptual level. The arguments of the process cannot be fully known, because each specific alternative can provide additional arguments. The alternatives can also provide nested processes.

The *Growth* process template from the *AquaticHierarchicalLibrary* in Example 3.9 is defined as:

```
template process Growth(pp : PrimaryProducer, ns : Nutrient<1, inf>) {
    processes:
        NutrientLimitation(pp, <n:ns>);
    equations:
        td(pp.conc) = pp.maxGrowthRate ∗ pp.nutrientLim ∗ pp.conc,
        td(<n:ns>.conc) = n.alpha ∗ pp.maxGrowthRate ∗ pp.nutrientLim ∗ pp.conc;
}
```

Every growth process created using the *Growth* template is nutrient limited. If we want to allow for unlimited growth, then we can substitute the definition of *Growth* with the following definitions:

```
template process Growth(pp : PrimaryProducer) {}
template process UnlimitedGrowth : Growth {
    equations:
        td(pp.conc) = pp.maxGrowthRate ∗ pp.conc;
```

Example 5.5: A model with incomplete top-level process.

```
incomplete model lakeModelIncompleteTop : AquaticHierarchicalLibrary;
entity phyto : PrimaryProducer {
    vars:
        conc{role: endogenous; initial: null},
        nutrientLim;
    consts:
        maxGrowthRate = 0.5,
}
entity phosphorus : Nutrient {
    vars:
        conc{role: endogenous; initial: null};
    consts:
        halfSaturation = null,
        alpha = 0.1;
}
entity nitrogen : Nutrient {
    vars:
        conc;
    consts:
        halfSaturation = null,
        alpha = 0.7;
}
process growth(phyto) : Growth {}
```

```
}
template process LimitedGrowth(ns : Nutrient<1, inf>) : Growth {
    processes:
        NutrientLimitation(pp, <n:ns>);
    equations:
        td(pp.conc) = pp.maxGrowthRate ∗ pp.nutrientLim ∗ pp.conc,
        td(<n:ns>.conc) = n.alpha ∗ pp.maxGrowthRate ∗ pp.nutrientLim ∗ pp.conc;
}
```

In this case, the *Growth* process is a conceptual process, with only one argument of type *PrimaryProducer*. There are two specific processes which provide alternatives for the *Growth* process. The first one is *UnlimitedGrowth* which defines unlimited exponential growth of phytoplankton, uninfluenced by any other external force. This alternative is typically not suitable for modeling growth, because in real-world ecosystems, the phytoplankton growth is highly dependent on available nutrients and environmental conditions. In certain scenarios, however, especially when experimenting with simulated data, it would be useful to consider it as a possible alternative. The second alternative, *LimitedGrowth*, is defined in the same way as the original *Growth* process from Example 3.9.

An incomplete model with a conceptual top-level process is presented in Example 5.5. The *growth* process uses the newly defined conceptual *Growth* template. The structure of this model is depicted in Figure 5.9. The *growth* process is depicted as a blank red square suggesting that it is a conceptual process. Only the *phyto* entity is specified as an argument and is therefore visually connected to the growth process.

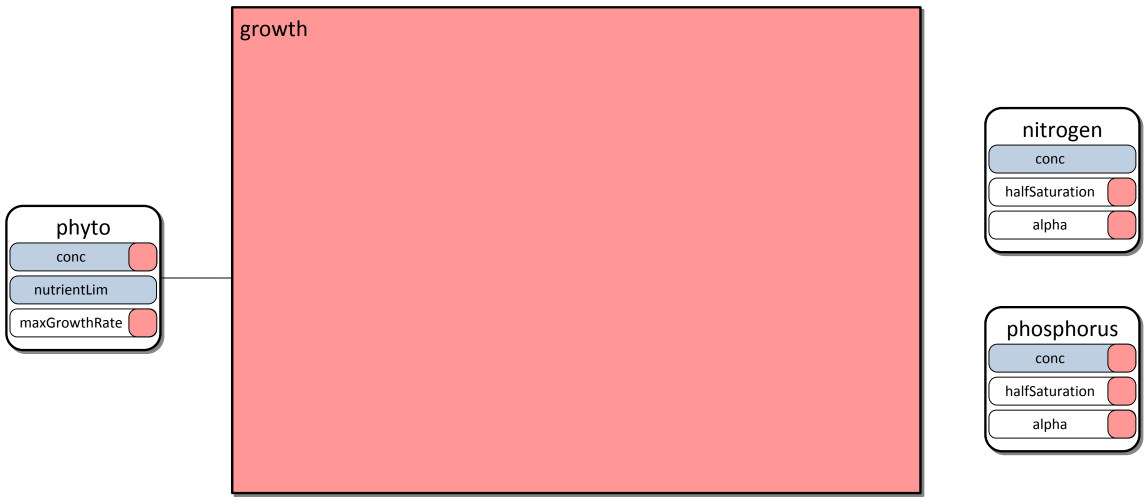


Figure 5.9: A model with incomplete top-level process types.

5.5 Inductive Process Modeling Task

Process-based modeling components form the basic substrate for inductive process modeling (IPM) methods. The aim of inductive process modeling is to find a suitable process-based model of an observed system, for which we have some background knowledge and some measured data. The model to be found should be complete.

The starting point of formulating the task of inductive process modeling is a dynamical system in which we are interested. We devise a library of domain knowledge which captures the available modeling knowledge about the class of such systems. We formalize our hypothesis in an incomplete process-based model of the observed system. The aspects of the system which are known are given and fixed in the incomplete model. The incomplete model also contains the aspects of the system which are not known for certain. Each such unknown aspect of the system is formalized as an incomplete specification, which also defines the possible solutions to resolve the incompleteness.

The other principal aspect of the dynamical system, in addition to the domain knowledge, are the available measurements of the system behavior. The measurements consist of two parts: measurements of the external (exogenous) variables and measurements of the observed state of the system. The exogenous variables are not modeled within the system, and we need to provide measurements for them in order to be able to simulate the model. The measurements of the observed state variables are used for evaluating the model.

In addition to background knowledge and measured data, we can provide an error function which is used to compare the simulated behavior of the model to the measured behavior of the system.

The outcome of inductive process modeling is a list of candidate models that satisfy the incomplete model and minimize the simulation error on the measured data. Each candidate model is a complete process-based model, i.e., it has both a complete structure and constant parameter values . All candidate models have different structure, yet they all satisfy the incomplete model.

We pose the IPM task as follows:
Given:

- a library of domain knowledge,
- an incomplete model,
- measured data and
- an error function.

Find:

- a ranked list of candidate models.

The IPM task can be decomposed into two subtasks: enumeration of structures and parameter estimation. The first subtask refers to enumerating all the candidate models that correspond to the given domain knowledge and incomplete model. Each model structure enumerated within the first task, has to go through the second subtask of estimating the proper values of the model parameters.

5.5.1 Model Structure Enumeration Subtask

The subtask of model structure enumeration revolves around the notion of *model completion*. We introduce a *completion* relation between two models, one incomplete model, which is being *completed* and one structurally complete model, which is the one that *completes*. In general terms, the structurally complete model is a completion of the incomplete model if it preserves all complete aspects of the incomplete model and substitutes all structural incomplete definitions with structurally complete ones without introducing unnecessary constructs.

Let $\widehat{M}$ be an incomplete model and $\overline{M}$ be a structurally complete model. We say that $\overline{M}$ is a completion of $\widehat{M}$ if all of the following hold:

1. the compartmental structures of $\widehat{M}$ and $\overline{M}$ are the same,
2. the entities of $\widehat{M}$ and $\overline{M}$ are the same,
3. all structurally complete processes in $\widehat{M}$ are also present in $\overline{M}$,
4. each structurally incomplete process in $\widehat{M}$ is completed in $\overline{M}$ in exactly one way,
5. $\overline{M}$ does not introduce any new top-level processes not present in $\widehat{M}$.

Statements 1., 2., and 3. imply that all complete aspects of the incomplete model must be maintained. Statement 4. demands that all structural incompleteness is removed. Finally, statement 5. prohibits adding superfluous processes to the complete model. The requirements from statements 1-5 can be formally expressed with the following propositions:

$$compartments^R(\overline{M}) = compartments^R(\widehat{M}) \tag{5.25a}$$

$$entities^R(\overline{M}) = entities^R(\widehat{M}) \tag{5.25b}$$

$$\left(\forall P \in processes^R(\widehat{M})\right) P \in complete^P(\widehat{M}) \implies P \in processes^R(\overline{M}) \tag{5.25c}$$

$$\begin{gathered}\left(\forall \widehat{P} \in processes^R(\widehat{M}\right) \widehat{P} \in incomplete^P(\widehat{M}) \implies \big(\exists! \\ P \in processes^R(\overline{M})\big)\, completion(P, \widehat{P})\end{gathered} \tag{5.25d}$$

$$\left|processes^{\mathrm{TOP}}(\widehat{M})\right| = \left|processes^{\mathrm{TOP}}(\overline{M})\right| \tag{5.25e}$$

With this notion of model completion, the model structure enumeration subtask then becomes the task of generating completions of the incomplete model given using the components from a library.

The model structure enumeration subtask is defined as:

Given:

- a library of domain knowledge and
- a structurally incomplete model.

Generate:

- a set of structurally complete candidate models.

5.5.2 Parameter Estimation Subtask

The second subtask in inductive process modeling is the estimation of the constant parameters. Starting with a structurally complete model, the task is to find suitable values for the constant parameters such that the discrepancy between the model's simulated behavior and the system's observed behavior is minimized. The model's behavior is obtained by numerical simulation of the model's equations, whereas the system's behavior is represented by the measurements of the observed state variables. The discrepancy between the two behaviors is measured by an error function.

The parameter estimation subtask is defined as:

Given:

- a structurally complete model with incomplete parameters,
- measured data and
- an error function.

Find:

- a complete model that minimizes the error function on the measured data.

Let $\overline{M}$ be a structurally complete model with unspecified parameters $\mathit{parameters}^{\ominus}(\overline{M})$. The state variables with unspecified initial values are denoted as $\mathit{state}^{\ominus}(\overline{M})$ and the constants with unspecified values are denoted as $\mathit{constants}^{\ominus}(\overline{M})$. The resulting complete model M includes specifications for all unspecified parameters which are in the allowed fitting range:

$$\big(\forall V \in \mathit{state}^{\ominus}(\overline{M})\big)\, \mathit{initial}|_M(V) \in \mathit{range}^F(V) \tag{5.26a}$$

$$\big(\forall C \in \mathit{constants}^{\ominus}(\overline{M})\big)\, \mathit{value}|_M(C) \in \mathit{range}^F(C) \tag{5.26b}$$

where $\mathit{initial}|_M$ and $\mathit{value}|_M$ respectively denote the definitions of the mappings *initial* and *value* within the model M.

6 ProBMoT

ProBMoT (**Pro**cess-**B**ased **Mo**deling **T**ool) is a methodology for solving the IPM task. It uses the process-based formalism for describing the domain knowledge organized in libraries. Each library is contained in a file called the *process-based library* file with the extension `.pbl`. It also uses the process-based formalism for expressing incomplete process-based models used as input and complete process-based models used as output. Both complete and incomplete models are contained in files called *process-based model* files with the extension `.pbm`.

The measurement data needed for simulation and evaluation of models is contained in a comma-separated (CSV) file which records all measured variables at successive time steps:

time	V_1	V_2	$\cdots$	V_m
t_0	v_{01}	v_{02}	$\cdots$	v_{0m}
t_1	v_{11}	v_{12}	$\cdots$	v_{1m}
t_2	v_{21}	v_{22}	$\cdots$	v_{2m}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
t_n	v_{n1}	v_{n2}	$\cdots$	v_{nm}

In addition, ProBMoT also utilizes an output specification. The output specification consists of output variables and output constants. The output variables are functions of all variables and constants within the model. The output constants are numerical parameters which can be used within the definition of output variables. The output constants, just like model constants, can be given a fixed value or can be fitted within some interval. The output variables essentially define what is observed as an output of the system.

In most scenarios, we are able to directly measure the state variables and therefore the output variables are simply a subset of the state variables, i.e., the observed state variables. In some scenarios, we are unable to directly measure a state variable and instead can only measure combinations of state variables like the total amount of several chemical compounds instead of the individual amounts. In these scenarios, the output variables are functions of the state variables which model the type of measurements we are able to perform.

The ProBMoT methodology proceeds in two stages, each solving one of the IPM subtasks: model structure enumeration and parameter estimation. The pipeline of ProBMoT is presented in Figure 6.1. The model structure enumeration stage takes as input an incomplete process-based model and a library of domain knowledge and enumerate all possible completions of the given incomplete model. All candidate model structures enter the parameter estimation stage. This stage utilizes the measurement data and output specification to produce complete process-based models with parameters that minimize the given error function. At the end, all candidate models are ranked according to the assigned error value.

6.1 Model Structure Enumeration Stage

For a given incomplete process-based model, ProBMoT enumerates all its completions, hence solving the model structure enumeration IPM subtask by generating all candidate model

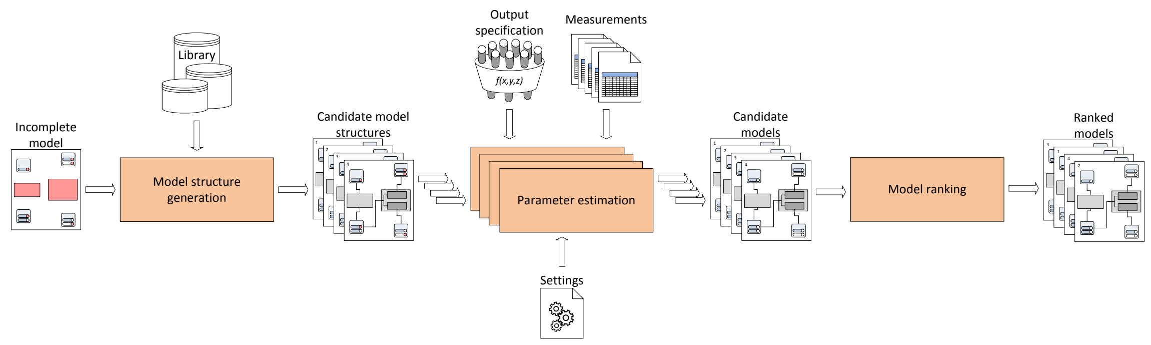


Figure 6.1: Overview of the ProBMoT pipeline

structures. The enumeration of model structures proceeds in three phases as depicted in Figure 6.2.

A structurally incomplete model enters the first phase called **top-level process resolution**. In this phase, ProBMoT substitutes all conceptual top-level processes with their specifications. The result of this phase is one or more, still structurally incomplete, models in which all top-level processes are specific process instances.

All models produced in the first phase enter the second phase called **argument binding**. Models in this phase are processed one at a time and independently of one another. Each unbound argument of a top-level process is assigned a set of entities which satisfies the lower and upper bound constraints of that argument. Each model taken as input in this phase results in one or more models. These models, in addition to having all only specific top-level processes, have bound process arguments. They can be still structurally incomplete, however, because they may contain nested processes which have a conceptual type.

The models produced in the second phase then enter the third phase called **nested process resolution**. The only structural incompleteness which is left by this point is conceptual nested processes. In this phase, each conceptual nested process is substituted with a process that is its completion. The completion of a nested process can have its own nested processes, which it turn can be conceptual. This phase is therefore recursive in nature.

In each phase, one or more structurally incomplete models enter and one or more model exit in which one type of incompleteness is removed. The last phase results with only structurally complete models.

6.1.1 Process Refinement

Let P be a process instance with a conceptual type $TP = template(P)$. Given a specific process type $TP' \in specifications(TP)$, we can construct a process instance P' such that $template(P') = TP'$ and all properties (arguments, constants, equations, and nested processes) of P are carried on to P'. The properties of P' must satisfy the following statements:

$$\left(\forall i \in \mathbb{N}_1^{|arguments^{\mathrm{T}}(TP)|}\right) arguments(P')_i = arguments(P)_i \tag{6.1a}$$

$$constants(P) \subset constants(P') \tag{6.1b}$$

$$\left(\forall i \in \mathbb{N}_1^{|equations(TP)|}\right) equations(P')_i = equations(P)_i \tag{6.1c}$$

$$\left(\forall i \in \mathbb{N}_1^{|processes^{\mathrm{T}}(TP)|}\right) processes(P')_i = processes(P)_i \tag{6.1d}$$

Besides those properties, TP' may contain additional arguments, constants, equations, and nested processes not present in TP. Therefore, P' has to contain proper instances for the additional arguments, constants and nested properties (equations in instance processes are inferred from the equation templates and arguments).

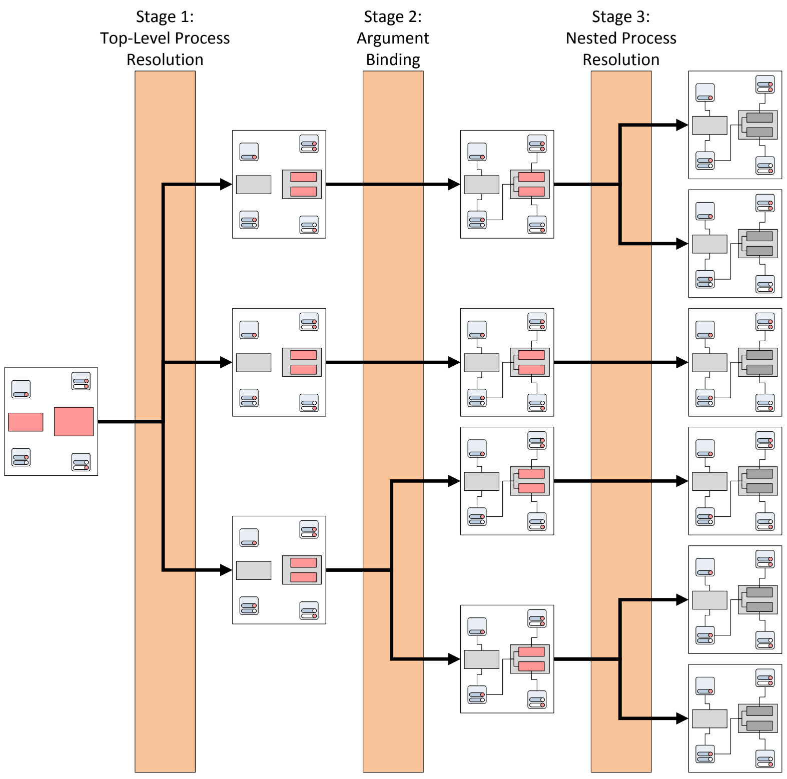


Figure 6.2: Enumeration of candidate model structures

We construct P' to be an incomplete process, and all properties not present in P are left unspecified. The newly introduced arguments which do not appear in P are unspecified:

$$\left(\forall i \in \mathbb{N}_{|arguments^{\mathrm{T}}(TP)|+1}^{|arguments^{\mathrm{T}}(TP')|}\right) arguments(P')_i = \text{undefined} \tag{6.2}$$

and their lower and upper bound include all possibilities, equivalent to the specification *[[], [all]]*:

$$\left(\forall i \in \mathbb{N}_{|arguments^{\mathrm{T}}(TP)|+1}^{|arguments^{\mathrm{T}}(TP')|}\right) bound^{\mathrm{L}}(P',i) = \varnothing \tag{6.3a}$$

$$\left(\forall i \in \mathbb{N}_{|arguments^{\mathrm{T}}(TP)|+1}^{|arguments^{\mathrm{T}}(TP')|}\right) \\ bound^{\mathrm{U}}(P',i) = \left\{\forall E \in entities^{\mathrm{R}}(M) \middle| ancestor^{\mathrm{E}}\left(arguments^{\mathrm{T}}(TP')_i, template(E)\right)\right\} \tag{6.3b}$$

The newly introduced constants which do not appear in P are also left with unspecified values:

$$\left(\forall C \in constants(P') \setminus constants(P)\right) value(C) = \text{undefined} \tag{6.4}$$

and their fitting ranges are taken to be the same as their range of allowed values:

$$\left(\forall C \in constants(P') \setminus constants(P)\right) range^{\mathrm{F}}(C) = range(template(C)) \tag{6.5}$$

We denote P' as $refinement(P,TP')$ and say that P' is a *refinement of P to type TP'*. P has a conceptual type and may contain other types of incompleteness (arguments and constant values). P' resolves the conceptual type incompleteness, but may add other types of incompleteness and contains all retains all other types of incompleteness that P has.

6.1.2 Resolution of Top-Level Processes

The resolution of top-level processes is the first phase of the enumeration of model structures. In this phase, the incomplete model that is part of the definition of the IPM task enters ProBMoT. ProBMoT then makes internal copies of this model and in each copy substitutes each conceptual top-level process with one of its refinements. All models created in this phase are structurally different, as different process refinement combinations are used in each model. The result of the first stage is the set of models created in this manner.

Algorithm 6.1 Resolution of top-level processes

```
1: procedure TLPRESOLUTION($M$) returns $MS$
2:     $MS \leftarrow \varnothing$
3:     $PS \leftarrow processes^{\text{TOP}}(M) \cap processes^{\ominus}(M)$        ▷ $PS = \{\widehat{P}_1, \ldots, \widehat{P}_n\}$
4:     $PS^{\text{SPEC}} \leftarrow \prod_{i=1}^{|PS|} specifications(template(\widehat{P}_i))$        ▷ Generate all specification combinations
5:     for all $P^{\text{SPEC}} \in PS^{\text{SPEC}}$ do
6:         $M' \leftarrow$ COPY($M$)
7:         for $i \leftarrow 1, |PS|$ do
8:             within $M'$ substitute $\widehat{P}_i$ with $refinement(\widehat{P}_i, P_i^{\text{SPEC}})$
9:         end for
10:        $MS \leftarrow MS \cup M'$
11:    end for
12:    return $MS$
13: end procedure
```

The procedure for generating models with resolved top-level processes is presented in Algorithm 6.1. After determining the set of incomplete top-level processes *PS*, ProBMoT generates all possible tuples of specifications to the incomplete top-level processes PS^{SPEC}. For each tuple of specifications P^{SPEC}, it constructs a model M' in which the processes from *PS* are substituted with their refinements. The result of this phase is the set of all models generated in this manner *MS*.

6.1.3 Argument Binding

The binding of process arguments is the second phase of the enumeration of model structures. Each model produced in the first phase passes trough this phase. ProBMoT makes internal copies of each model and in each copy binds each unbound argument to a set of entities. The models created in this phase differ between themselves in the bindings of the argument, because ProBMoT choses different combinations of argument bindings in each model. The result of the second stage is a set of models in which all top-level processes are resolved and all process arguments are bound to particular entity sets.

Algorithm 6.2 presents the recipe for generating models with bound arguments. The ARGUMENTSETS procedure generates the allowed bindings for each unbound argument. It defines the function *AS* which maps each unbound argument $pos \in positions^{\ominus}(M)$ to a set of allowed argument combinations for that parameter *comb*. First, the *mandatory* and *optional* sets of entities and the set of possible cardinalities of the argument *cardinalities* are determined. Then, for each possible cardinality $card \in cardinalities$, the set of combinations of the *optional* entities choose *card* elements. All combinations of allowed cardinalities are gathered in (*comb*) and assigned to *AS*(*pos*).

The ARGUMENTBINDING procedure works in a similar way to the TLPRESOLUTION procedure from phase one. The possible argument bindings for each argument are stored in the *AS* mapping and all combinations of arguments binding for the model *M* are constructed

Algorithm 6.2 Argument binding

```
1: procedure ArgumentSets($M$) returns $AS(\cdot)$
2:     $AS \leftarrow \varnothing$                                          ▷ Empty mapping
3:     for all $pos \in positions^{\ominus}(M)$ do                  ▷ For all undefined arguments
4:         $mandatory \leftarrow bound^{U}(pos)$                              ▷ $pos = (P, i)$
5:         $optional \leftarrow bound^{U}(pos) \setminus bound^{L}(pos)$
6:         $cardinalities \leftarrow arguments^{C}(template(P)) \cap \mathbb{N}_0^{|optional|}$
7:         $comb \leftarrow \varnothing$
8:         for all $card \in cardinalities$ do
9:             $comb \leftarrow comb \cup$ Combinations($optional, card$)
10:        end for
11:        $AS(pos) = comb$                 ▷ All combinations for the undefined argument
12:    end for
13:    return $AS$
14: end procedure
15: procedure ArgumentBinding($M$) returns $MS$
16:    $MS \leftarrow \varnothing$
17:    $AS \leftarrow$ ArgumentSets($M$)
18:    $binding^{\text{COMB}} \leftarrow \prod_{pos \in positions^{\ominus}(M)} AS(pos)$
19:    for all $b^{\text{COMB}} \in binding^{\text{COMB}}$ do
20:        $M' \leftarrow$ Copy($M$)
21:        for all $pos \in positions^{\ominus}(M)$ do
22:            within $M'$ bind argument $pos$ to $b^{\text{COMB}}_{pos}$
23:        end for
24:        $MS \leftarrow MS \cup M'$
25:    end for
26:    return $MS$
27: end procedure
```

in *binding*$^{\text{COMB}}$. For each combination of argument bindings b^{COMB} a model M' with that assignment of arguments is created and added to the set of resulting models *MS*.

6.1.4 Resolution of Nested Processes

The third phase of the enumeration of model structures is the resolution of nested processes. The models produced in the second phase carry on this phase. The nested process resolution is conceptually similar to the first phase—resolution of top-level processes, where ProBMoT substitutes all conceptual top-level processes with one of their refinements. Each process refinement, however, may introduce new nested processes. The result will then be a model which does not contain conceptual top-level processes, but may contain conceptual nested processes. The procedure for resolving the newly added conceptual nested processes is then the same, creating a recursive algorithm.

Algorithm 6.3 Resolution of nested processes

```
1: procedure NPRESOLUTION($M$) returns $MS$
2:     $MS \leftarrow \varnothing$
3:     $PS \leftarrow processes^{\ominus}(M)$                      ▷ $PS = \{\widehat{P}_1, \ldots, \widehat{P}_n\}$
4:     $PS^{\text{SPEC}} \leftarrow \prod_{i=1}^{|PS|} specifications(template(\widehat{P}_i))$     ▷ Generate all specification combinations
5:     for all $P^{\text{SPEC}} \in PS^{\text{SPEC}}$ do
6:         $M' \leftarrow$ COPY($M$)
7:         for $i \leftarrow 1, |PS|$ do
8:             within $M'$ substitute $\widehat{P}_i$ with $refinement(\widehat{P}_i, P_i^{\text{SPEC}})$
9:         end for
10:        if $processes^{\ominus}(M') = \varnothing$ then
11:            $MS \leftarrow MS \cup M'$
12:        else
13:            $MS \leftarrow MS \cup$ NPRESOLUTION($M'$)
14:        end if
15:    end for
16:    return $MS$
17: end procedure
```

The procedure for generating models with resolved nested processes is presented in Algorithm 6.3. ProBMoT determines the conceptual nested processes *PS* and generates the possible tuples of specifications to the conceptual nested processes PS^{SPEC}. For each tuple of specifications P^{SPEC}, it constructs a model M' in which the processes from *PS* are substituted with their refinements. The refinements of conceptual processes $refinement(\widehat{P}_i, P_i^{\text{SPEC}})$ may introduce new conceptual nested processes, thus M' may still be a structurally incomplete model. If there are still conceptual processes within M' they are recursively eliminated with NPRESOLUTION(M'). The result of this phase is the a set of structurally complete models.

6.1.5 Finiteness of the Space of Candidate Model Structures

The process-based formalism is a powerful language that enables construction of structurally complex models. A structurally incomplete model can contain different types of incompleteness and the interplay of the different incompleteness types creates a complex space of candidate model structures. A key feature of ProBMoT's structure enumeration stage is that it tackles the different types of incompleteness one by one. Each type of incompleteness it addresses by one phase of the structure enumeration stage. The responsibility of each phase is to generate all candidate model structures that will resolve that type of

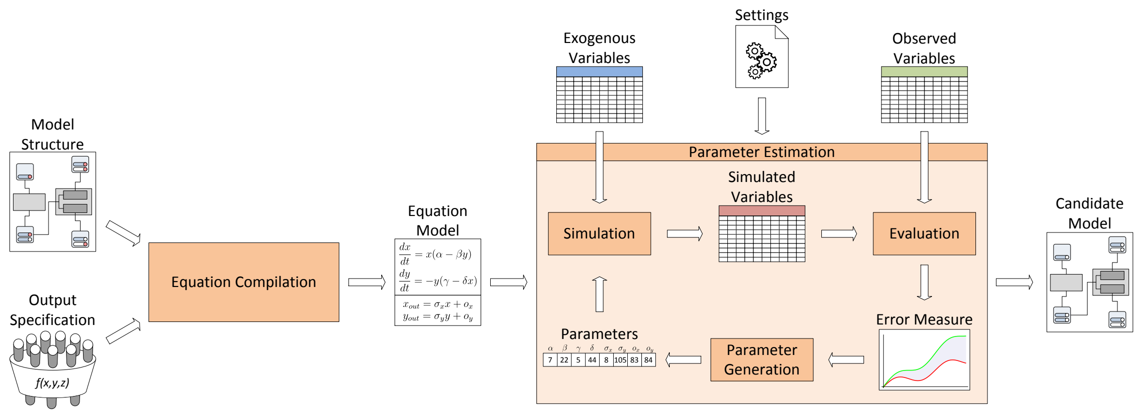


Figure 6.3: Estimation of parameters of a single candidate model structure

incompleteness. This layered approach ensures that no incompleteness is left unaddressed and all possible candidates are considered.

The space of all candidate model structures is finite. This comes as a result of the finite number of candidate models generated in each phase. We therefore, move to show that each phase generates a finite number of models.

In the first phase, only the top-level processes are considered. As there is a finite number of top-level processes and a finite number of process templates to choose from, ProBMoT generates a finite number of candidate models.

In the second phase, the argument bindings of the top-level processes are considered. As the number of top-level processes and the number of arguments per process are finite, the total number of arguments in the model is also finite. For each unbound argument, ProBMoT considers all subsets of entities of the suitable types. The number of entities in the model is finite and as a result, the number of possible argument bindings has to be finite.

In the third phase, the nested processes are considered. As the resolution of nested processes can introduce additional nested processes, ProBMoT resolves nested processes recursively. In order to ensure that a recursive procedure is finite, we have to take appropriate measures. In this case, the finiteness of the recursion is a result of the fact that the nested processes can not be cyclically nested and moreover, the types of nested processes can not be cyclically nested.

6.2 Parameter Estimation Stage

6.2.1 Model Output

The model can be expressed as a set of ordinary differential equations:

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}(t), \mathbf{u}(t)) \tag{6.6}$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ represents the complete internal state of the system and the exogenous variables $\mathbf{u}(t) \in \mathbb{R}^k$ capture the influence of the environment. Observing the system means that we monitor a set of variables $\mathbf{y}(t) \in \mathbb{R}^m$ that depend on the time t, the system's internal state $\mathbf{x}(t)$, and the external input $\mathbf{u}(t)$:

$$\mathbf{y} = \mathbf{h}(t, \mathbf{x}(t), \mathbf{u}(t)) \tag{6.7}$$

The simplest observational scenario is where we can measure a subset of the state variables, i.e., $\mathbf{y} = (x_{i_1}(t), \dots, x_{i_m}(t)), \quad I = \{i_1, i_2, \dots, i_m\} \subset \{1, 2, \dots, n\}$. These variables $\mathbf{x}_I$ are the observed variables whereas the rest of the state variables $\mathbf{x}_{\mathbb{N}_1^n \setminus I}$ are the hidden (or unobserved) variables.

In systems which involve many state variables, it is often expensive or even not technologically possible to measure all state variables. Therefore, for large and complex systems it is commons to have only partial observability. Another origin of such scenario is in the case of data repurposing, when the acquisition of the data is performed before the modeling scenario is known and it is not possible to revise the experiment.

In some cases, a single output variable can depend on several state variables. A typical situation in modeling of ecosystems is when the measurement technique cannot distinguish between two or more state variables. An example is an aquatic ecosystem in which there are two types of phytoplankton (diatoms and dinoflagellates), modeled as separate entities (*phytoDia* and *phytoDino* respectively), each involved in their own trophic processes. The measured data, however, may not contain separate measurements, but instead contain cumulative concentration ($\mathit{phytoMeasured} = \mathit{phytoDia} + \mathit{phytoDino}$).

Within ProBMoT, the observability scenario is modeled with an output specification. The output specification is a set of output constants, which can be fixed or unspecified, and a set of output variables, which are modeled with algebraic equations.

In particular, let $\mathcal{O}$ be the set of all output specification. Then, the elements of $\mathcal{O}$ are defined with the following mappings:

$$\mathit{constants} : \mathcal{O} \to \mathscr{P}(\mathcal{OC}) \tag{6.8a}$$
$$\mathit{variables} : \mathcal{O} \to \mathscr{P}(\mathcal{OV}) \tag{6.8b}$$

where $\mathcal{OC}$ is the set of output constants and $\mathcal{OV}$ is the set of output variables.

The elements of $\mathcal{OC}$ are defined with a single mapping:

$$\mathit{value} : \mathcal{OC} \to \mathbb{R} \tag{6.9}$$

which captures the value of the output constant.

Each output constant, similarly to a model constant, can be specified or unspecified. We therefore partition the set of output constants $\mathit{constants}(O)$ into a set of specified output constants $\mathit{constants}^{\oplus}(O)$ and a set of unspecified output constants $\mathit{constants}^{\ominus}(O)$, such that $\mathit{constants}^{\oplus}(O) \cap \mathit{constants}^{\ominus}(O) = \varnothing$ and $\mathit{constants}^{\oplus}(O) \cup \mathit{constants}^{\ominus}(O) = \mathit{constants}(O)$. The *value* mapping is defined only for the elements of $\mathit{constants}^{\oplus}(O)$:

$$\mathit{value}(C) = \begin{cases} r \in \mathbb{R} & C \in \mathit{constants}^{\oplus}(O) \\ \text{undefined} & C \in \mathit{constants}^{\ominus}(O) \end{cases} \tag{6.10}$$

The elements of $\mathcal{OV}$ are defined with the mappings:

$$\mathit{value} : \mathcal{OV} \to (\mathbb{R} \to \mathbb{R}) \tag{6.11a}$$
$$\mathit{function} : \mathcal{OV} \to (\mathbb{R}^n \to \mathbb{R}) \tag{6.11b}$$
$$\mathit{inputs} : \mathcal{OV} \to (\mathbb{N}_1^n \to \mathcal{V} \cup \mathcal{C} \cup \mathcal{OC}) \tag{6.11c}$$

The *value* mapping represents the value of the variable trough time and is therefore modeled as a real function. Each output variable has an associated equation, the functional form of which is represented with the *function* mapping. The *inputs* mapping represents the inputs that appear in the equation for the particular output variable. Each input can be a model variable or constant or output constant.

6.2.2 Equation Compilation

The task of the equation compilation stage is, given a completely specified process-based model and an output specification, to translate them into the equivalent equations. The result of the equation compilation is a list of ordinary differential equations and a list of algebraic equations.

The first step is to gather all equation fragments that influence an endogenous variables into a single equations. For each endogenous variable $V \in \mathit{endogenous}(M)$, we designate by $\mathit{equations}(V)$ the set of all equations that influence V:

$$\mathit{equations}(V) = \{Q \in \mathcal{Q} | \mathit{lhs}(Q) = V\} \tag{6.12}$$

We assume that the set $\mathit{equations}(V)$ has some (arbitrary) ordering, such that:

$$\mathit{equations}(V) = \{Q_1, Q_2, \ldots, Q_k\} \tag{6.13}$$

Each equation Q_i has a functional form denoted by $\mathit{function}(Q_i)$ and a specification of the input arguments $\mathit{inputs}(Q_i)$. We define the mapping $\mathit{function}^{\mathrm{EQ}}$ that is compositions of all *function* mappings that influence a single variable V:

$$\mathit{function}^{\mathrm{EQ}}(V) = \mathit{aggregation}(V)(\mathit{function}(Q_1), \ldots, \mathit{function}(Q_k)) \tag{6.14}$$

We also define the mapping $\mathit{inputs}^{\mathrm{EQ}}$ that is a concatenation of all *inputs* mappings that influence a single variable V composed with Algorithm 6.4.

Algorithm 6.4 Generating the *inputs* mapping

```
1: procedure GenerateInputs($V$) returns $inputs^{EQ}(V)$
2:     $inputs^{EQ}(V) \leftarrow \varnothing$
3:     $k \leftarrow 0$
4:     $QS \leftarrow equations(V)$                                  $\triangleright\ QS = \{Q_1, \ldots, Q_n\}$
5:     for all $Q \in QS$ do
6:         for all $(i, A) \in inputs(Q)$ do                          $\triangleright\ A \in \mathcal{V} \cup \mathcal{C}$
7:             $inputs^{EQ}(V) \leftarrow inputs^{EQ}(V) \cup (k + i, A)$
8:         end for
9:         $k \leftarrow k + |inputs(Q)|$
10:    end for
11:    return $inputs^{EQ}(V)$
12: end procedure
```

ProBMoT performs a two-step simulation. The first step is a numerical simulation of a set of ordinary differential equations. The second step is a simple computation of algebraic functions which involve equations from the model and output equations.

The model output is the ultimate result of the simulation of a process-based model. The output contains the most important state variables or functions thereof. Depending on the scenario, the user may be interested in one or only a few state variables, which correspond to one part or aspect of the modeled system. Suitable output equations are then provided in the output specification that reflect the interest of the user.

In such a setup, not all equations are necessary to compute the values of the output variables. The equations which are not needed in order to produce simulations for the output variables are not computed. ProBMoT determines the necessary variables and constants for the algebraic and differential simulation separately.

First, starting from the output variables and their associated equations, it determines the variables and constants needed for the algebraic simulation (Algorithm 6.5). It follows all endogenous variables that have associated algebraic equations and traverses all variables that occur in those equations. The algorithm stops the exploration when it reaches a state or exogenous variable.

Second, starting from the state variables required for the algebraic simulation, ProBMoT does another traversal on the equations graph, this time marking the variables and constants needed for the differential simulation (Algorithm 6.6). In this pass, it follows all endogenous variables, traversing their associated algebraic and differential equations, stopping when it reaches exogenous variables.

Algorithm 6.5 Determining the needed equations for the algebraic simulation

```
1: procedure NEEDEDALGEBRAIC($M$,$O$) returns ($needed^{V}_{ALG}$, $needed^{C}_{ALG}$)
2:     $pending \leftarrow$ empty Queue
3:     $traversed \leftarrow \varnothing$
4:     $needed^{C}_{ALG} \leftarrow \varnothing$
5:     $needed^{V}_{ALG} \leftarrow \varnothing$
6:     for all $OV \in variables(O)$ do
7:         for all variables $V$ that appear in $inputs(OV)$ do
8:             $pending$.enqueue($V$)
9:             $needed^{V}_{ALG} \leftarrow needed^{V}_{ALG} \cup \{V\}$
10:        end for
11:        for all constants $C$ that appear in $inputs(OV)$ do
12:            $needed^{C}_{ALG} \leftarrow needed^{C}_{ALG} \cup \{C\}$
13:        end for
14:    end for
15:    while $pending$ is not empty do
16:        $\bar{V} \leftarrow pending$.dequeue
17:        if $\bar{V} \notin traversed$ then
18:            $traversed \leftarrow traversed \cup \{\bar{V}\}$
19:            if $auxiliary(\bar{V}) = \text{true}$ then
20:                for all variables $V$ that appear in $inputs(\bar{V})$ do
21:                    $pending$.enqueue($V$)
22:                    $needed^{V}_{ALG} \leftarrow needed^{V}_{ALG} \cup \{V\}$
23:                end for
24:                for all constants $C$ that appear in $inputs(\bar{V})$ do
25:                    $needed^{C}_{ALG} \leftarrow needed^{C}_{ALG} \cup \{C\}$
26:                end for
27:            end if
28:        end if
29:    end while
30:    return ($needed^{V}_{ALG}$, $needed^{C}_{ALG}$)
31: end procedure
```

Algorithm 6.6 Determining the needed equations for the differential simulation

```
 1: procedure NeededDifferential($M$,$needed^{V}_{ALG}$) returns ($needed^{V}_{DIF}$,$needed^{C}_{DIF}$)
 2:     $pending \leftarrow$ empty Queue
 3:     $traversed \leftarrow \varnothing$
 4:     $needed^{C}_{DIF} \leftarrow \varnothing$
 5:     $needed^{V}_{DIF} \leftarrow \varnothing$
 6:     for all $V \in needed^{V}_{ALG}$ do
 7:         if $state(V) = \text{true}$ then
 8:             $pending.\text{enqueue}(V)$
 9:             $needed^{V} \leftarrow needed^{V} \cup \{V\}$
10:         end if
11:     end for
12:     while $pending$ is not empty do
13:         $\bar{V} \leftarrow pending.\text{dequeue}$
14:         if $\bar{V} \notin traversed$ then
15:             $traversed \leftarrow traversed \cup \{\bar{V}\}$
16:             if $endogenous(\bar{V}) = \text{true}$ then
17:                 for all variables $V$ that appear in $inputs(\bar{V})$ do
18:                     $pending.\text{enqueue}(V)$
19:                     $needed^{V}_{DIF} \leftarrow needed^{V}_{DIF} \cup \{V\}$
20:                 end for
21:                 for all constants $C$ that appear in $inputs(\bar{V})$ do
22:                     $needed^{C}_{DIF} \leftarrow needed^{C}_{DIF} \cup \{C\}$
23:                 end for
24:             end if
25:         end if
26:     end while
27:     return ($needed^{V}_{DIF}$,$needed^{C}_{DIF}$)
28: end procedure
```

Once the set of algebraic and differential equations necessary to produce the required output are determined, the next step is to sort those equations. The sorting procedure for the equations for the differential and algebraic simulations is carried out separately, even though it is very similar. The sorting procedure is based on the most commonly used algorithm for topological sorting known as *Kahn's algorithm* (Kahn, 1962).

Algorithm 6.7 Sorting the equations needed for differential simulation

```
1: procedure SortDifferential($needed^{V}_{DIF}$) returns $L$
2:     $L \leftarrow$ empty List                    ▷ $L$ will contain the ordered variables
3:     $S \leftarrow$ empty Queue          ▷ $S$ contains the variables with no input dependencies
4:     $remaining \leftarrow \varnothing$      ▷ $remaining$ will contain the set of all input dependencies
5:     for all $V \in needed^{V}_{DIF}$ do
6:         if $endogenous(V) = \text{true}$ then
7:             for all endogenous variables $\bar{V}$ that appear in $inputs(V)$ do
8:                 $remaining \leftarrow remaining \cup \{(V, \bar{V})\}$
9:             end for
10:        end if
11:    end for
12:    for all $V \in needed^{V}_{DIF}$ do
13:        if $endogenous(V) = \text{true} \wedge inputs(V) = \varnothing$ then
14:            $S \leftarrow S \cup \{V\}$
15:        end if
16:    end for
17:    while $S$ is not empty do
18:        $V \leftarrow S.\text{dequeue}$
19:        insert $V$ into $L$
20:        for all endogenous variables $\bar{V}$ that appear in $inputs(V)$ do
21:            remove $(V, \bar{V})$ from $remaining$
22:            if there are no $\tilde{V}$ such that $(\tilde{V}, \bar{V}) \in remaining$ then
23:                $S.\text{enqueue}(\bar{V})$
24:            end if
25:        end for
26:    end while
27:    if $remaining = \varnothing$ then
28:        return $L$
29:    else
30:        raise Exception(cyclic dependency detected)
31:    end if
32: end procedure
```

Each endogenous variable takes the form of a node in a graph. The links between the nodes are the memberships of the variables in the equations associated with other variables. Kahn's algorithm starts with the set of nodes with no input dependencies, i.e., the endogenous variables that are not influenced by any equation, but only by exogenous variables. Each such endogenous variable appears on the left-hand side of an exactly one equation. All endogenous variables with no input dependencies are inserted into the resulting ordered list and removed from the graph. The order in which they are inserted is not important because they are independent of each other. The algorithm then proceeds iteratively. In each iteration it finds the set of all endogenous variables which have no input dependencies (because they were removed in the previous iteration) and adds them to the resulting list. In the end, the algorithm should remove all variables from the graph. In case there are variables of which none has an empty set of input dependencies that signalizes a cyclic dependency between the variables. In such a case, ProBMoT terminates and returns an error indicating that it has encountered a cyclic dependency in the equations. The result of the topological sort is an ordered list of endogenous variables and represents the order in which the associated equations should be executed.

The result of the topological sort is not unique, since it represents a partial ordering. Note however, that any topologically sorted ordering of the equations is acceptable because they all preserve the dependencies between the variables and equations.

The procedure for sorting the equations needed for the differential simulation (Algorithm 6.7) and the procedure for sorting the equations needed for the algebraic simulation (Algorithm 6.8) are very similar. Operating on the set of needed equations, they iteratively remove equations that do not have input dependencies from the equation graph and add them to a list which tracks the resulting ordering. The key difference is that the sets of equations on which the procedures operate are different ($needed^{\mathrm{V}}_{\mathrm{ALG}}$ and $needed^{\mathrm{V}}_{\mathrm{DIF}}$ respectively for the algebraic and differential simulation). The other main difference is that the within the differential simulation, all endogenous equations including both differential and algebraic are taken as links in the equation graph, whereas within the algebraic simulation only algebraic equations are considered when traversing the equation graph.

Algorithm 6.8 Sorting the equations needed for algebraic simulations

```
1: procedure SortAlgebraic($needed^{\mathrm{V}}_{\mathrm{ALG}}$) returns $L$
2:     $L \leftarrow$ empty List
3:     $S \leftarrow$ empty Queue
4:     $remaining \leftarrow \varnothing$
5:     for all $V \in needed^{\mathrm{V}}_{\mathrm{ALG}}$ do
6:         if $auxiliary(V) = \mathrm{true}$ then
7:             for all auxiliary variables $\bar{V}$ that appear in $inputsV$ do
8:                 $remaining \leftarrow remaining \cup \{(V, \bar{V})\}$
9:             end for
10:        end if
11:    end for
12:    for all $V \in needed^{\mathrm{V}}_{\mathrm{ALG}}$ do
13:        if $auxiliary(V) = \mathrm{true} \wedge inputs(V) = \varnothing$ then
14:            $S \leftarrow S \cup \{V\}$
15:        end if
16:    end for
17:    while $S$ is not empty do
18:        $V \leftarrow S$.dequeue
19:        insert $V$ into $L$
20:        for all auxiliary variables $\bar{V}$ that appear in $inputs(V)$ do
21:            remove $(V, \bar{V})$ from $remaining$
22:            if there are no $\tilde{V}$ such that $(\tilde{V}, \bar{V}) \in remaining$ then
23:                $S$.enqueue($\bar{V}$)
24:            end if
25:        end for
26:    end while
27:    if $remaining = \varnothing$ then
28:        return $L$
29:    else
30:        raise Exception(cyclic dependency detected)
31:    end if
32: end procedure
```

6.2.3 Model Simulation

At this stage in the pipeline, ProBMoT has determined the set of state and exogenous variables needed in the differential and algebraic simulations. To streamline the notation in this section, we designate those sets as vectors with an arbitrary ordering: $\overline{X}$—the vector of state variables needed for the differential simulation, $\overline{U}$—the vector of exogenous variables needed for the differential simulation, $\widehat{X}$—the vector of state variables needed for the algebraic simulation, and $\widehat{U}$—the vector exogenous variables needed for the algebraic simulation.

In the differential simulation, given a time interval $T = [t_0, t_N]$, we want to determine the values of the state variables $\overline{X}$ on T, i.e., we want to determine the mapping $\mathit{values}|_T(\overline{X})$. During the simulation, the values of the exogenous variables $\mathit{values}|_T(\overline{U})$, as well as the initial values of the state variables $\mathit{initial}(\overline{X})$ are already known. At this point, the functional form of the equations associated with the state variables are given by $\mathit{function}^{\mathrm{EQ}}(\overline{X})$.

Substituting conventional notations for the state variables $\bar{\mathbf{x}} = \mathit{values}|_T(\overline{X})$, the exogenous variables $\bar{\mathbf{u}} = \mathit{values}|_T(\overline{U})$, the equation structure $\mathbf{f} = \mathit{function}^{\mathrm{EQ}}(\overline{X})$, and the initial values $\bar{\mathbf{x}}_0 = \mathit{initial}(\overline{X})$,we arrive at the standard formulation of a system of ordinary differential equations:

$$\frac{d}{dt}\bar{\mathbf{x}} = \mathbf{f}(t, \bar{\mathbf{x}}(t), \bar{\mathbf{u}}(t)) \tag{6.15a}$$

$$\bar{\mathbf{x}}(t_0) = \bar{\mathbf{x}}_0 \tag{6.15b}$$

The formulation given in Equation (6.15) is the standard definition of the *initial value problem* (IVP) of ODEs (Atkinson, 1989). To find the values of the functions $\bar{\mathbf{x}}$ means to numerically solve the initial value problem. Methods that solve the IVP, discretize the integration interval T into a series of time points $t_0, t_1, \ldots, t_N$ and solve a set of difference equations at each time point (Gupta et al., 1985).

ProBMoT uses CVODE for numerically solving ODEs and in particular relies on BDF in combination with a Krylov method (Saad and Schultz, 1986) for the linear solver within the Newton iterations.

The set of output algebraic equations in the form:

$$\mathbf{y} = \mathbf{h}(t, \mathbf{x}(t), \mathbf{u}(t)) \tag{6.16}$$

where $\mathbf{x}$ is the vector of needed state variables, $\mathbf{u}$ is the vector of needed exogenous variables and $\mathbf{y}$ is the vector of output variables.

6.2.4 Measuring Model Performance

Once the output variables are simulated, we obtain the output behavior of the model. We judge the performance of the model based on how well this behavior corresponds to the observed behavior of the output variables, typically acquired by experimental measurements.

The value of each output variable is defined as a function $\mathit{value}(OV) : \mathbb{R} \to \mathbb{R}$. In a particular observational scenario, however, the domain of the function is restricted to time interval T, so the mapping becomes: $T \to \mathbb{R}$. The values of all output variables $\mathit{variables}(O)$ are defined as such functions, thus making the behavior of the model output a multivariate time series: $T \to \mathbb{R}^n$, where n is the number of output variables.

In practice, the measurements, as well as the simulated output variables, are not continuous values, but presented at discrete time points. Let $\overline{T}$ be the discretization of the time interval T given as $\overline{T} = [t_0, t_1, \ldots, t_n]$. Let $\mathbf{y} = [y_1, \ldots, y_m]$ be the vector of output variables. The measurements of the output variables form a matrix $\overline{\mathbf{Y}} = [\overline{Y}_1, \ldots, \overline{Y}_m]$, where each $\overline{Y}_j = [\bar{y}_{0j}, \ldots, \bar{y}_{nj}]^\intercal$ is a column vector of the measurements for the output variable y_j at time points $t_0, \ldots, t_n$. The simulated output variables produce a similar matrix $\widehat{\mathbf{Y}} = [\widehat{Y}_1, \ldots, \widehat{Y}_m]$, where each $\widehat{Y}_j = [\widehat{y}_{0j}, \ldots, \widehat{y}_{nj}]^\intercal$ is a column vector of the simulated values for the output variable y_j at time points $t_0, \ldots, t_n$.

We represent visually the observed and simulated output behavior with the following scheme:

time	$\bar{Y}_1$	$\bar{Y}_2$	$\cdots$	$\bar{Y}_m$	$\widehat{Y}_1$	$\widehat{Y}_2$	$\cdots$	$\widehat{Y}_m$
t_0	$\bar{y}_{01}$	$\bar{y}_{02}$	$\cdots$	$\bar{y}_{0m}$	$\widehat{y}_{01}$	$\widehat{y}_{02}$	$\cdots$	$\widehat{y}_{0m}$
t_1	$\bar{y}_{11}$	$\bar{y}_{12}$	$\cdots$	$\bar{y}_{1m}$	$\widehat{y}_{11}$	$\widehat{y}_{12}$	$\cdots$	$\widehat{y}_{1m}$
t_2	$\bar{y}_{21}$	$\bar{y}_{22}$	$\cdots$	$\bar{y}_{2m}$	$\widehat{y}_{21}$	$\widehat{y}_{22}$	$\cdots$	$\widehat{y}_{2m}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
t_n	$\bar{y}_{n1}$	$\bar{y}_{n2}$	$\cdots$	$\bar{y}_{nm}$	$\widehat{y}_{n1}$	$\widehat{y}_{n2}$	$\cdots$	$\widehat{y}_{nm}$

The task of evaluating the performance of the model is translated to measuring how well these two m-variate time-series match each other. In the simplest scenario where we have only one output variable, the problem reduces to measuring the discrepancy between two single-variable time-series.

We use the maximum-likelihood estimator introduced by R.A.Fisher in 1912 (Pfanzagl, 1994) that maximizes the probability of observing the given data if a given model is chosen. The likelihood function depends on the probability of the measurements in the data set. Assuming that the measurements follow independent normal distributions with constant variance, the maximum-likelihood parameter estimation maps into a nonlinear least-squares estimation of the parameters, which minimizes the sum of squared errors between the observed values and the values predicted by the model. Due to its intuitive appeal and simplicity, least-squares estimation is commonly used for parameter estimation in nonlinear models.

The most basic form of least-squares estimation uses the sum of squared errors (*SSE*) for expressing the model error, which sums up the squares of the differences between the measured values $\bar{y}_i$ and the the predicted values $\widehat{y}_i$, at each time point i.

$$SSE = \sum_{i=0}^{n} (\bar{y}_i - \widehat{y}_i)^2 \tag{6.17}$$

Root mean squared error (*RMSE*) divides the *SSE* by the number of time points $n+1$ and takes the square root, making the measurement units and scale comparable to the ones of the observed output variable.

$$RMSE = \sqrt{\frac{1}{n+1} SSE} = \sqrt{\frac{1}{n+1} \sum_{i=0}^{n} (\bar{y}_i - \widehat{y}_i)^2} \tag{6.18}$$

Root relative squared error (*RRSE*) is the square root of the total squared error made relative to what the error would have been if the prediction had been the average of the absolute value. *RRSE* is expressed in percentages compared to the average as a predictor:

$$RRSE = \sqrt{\frac{\sum_{i=0}^{n} (\bar{y}_i - \widehat{y}_i)^2}{\sum_{i=0}^{n} (\bar{y}_i - \tilde{y})^2}} \tag{6.19}$$

where $\tilde{y}$ is the average of the measured data.

We decompose the error of the output vector $\mathbf{y}$ into the errors of each output variable y_i and aggregate them into a single error which we attribute to $\mathbf{y}$. ProBMoT allows us to specify arbitrary error measures for the output variables y_i as well as arbitrary aggregation schemes. It is important to keep in mind that if we use error measures that do not produce values on the same scale for all output variables, some output variables may take over.

The default implementation of an error measure for the output vector is a sum of the *RRSE* values for all output variables.

$$RRSE_{\text{TOTAL}} = \sum_{j=1}^{m} \sqrt{\frac{\sum_{i=0}^{n} (\bar{y}_{ij} - \widehat{y}_{ij})^2}{\sum_{i=0}^{n} (\bar{y}_{ij} - \tilde{y}_j)^2}} \tag{6.20}$$

6.2.5 Parameter Estimation

Given a structurally complete model $\boldsymbol{M}$, we denote with $\boldsymbol{parameters}(\boldsymbol{M})$ the set of all model parameters including initial values and constants. Among them, we denote the specified parameters with $\boldsymbol{parameters}^{\oplus}(\boldsymbol{M})$ and the unspecified with $\boldsymbol{parameters}^{\ominus}(\boldsymbol{M})$. Additionally, given an output specification $\boldsymbol{O}$, we denote with $\boldsymbol{constants}(\boldsymbol{O})$ the output parameters, including the specified $\boldsymbol{constants}^{\oplus}(\boldsymbol{O})$ and the unspecified $\boldsymbol{constants}^{\ominus}(\boldsymbol{O})$.

The task of the parameter estimation is to find the most suitable values for $\boldsymbol{parameters}^{\ominus}(\boldsymbol{M})$ and $\boldsymbol{constants}^{\ominus}(\boldsymbol{O})$, hence completing the ***values*** and ***initial*** mappings.

Parameter estimation leads to challenging optimization tasks that typically require advanced meta-heuristic approaches for global optimization, such as evolutionary or swarm-based methods. Ecological models are typically nonlinear and have many parameters; the measurements are sparse and imperfect due to noise. All these constraints can lead to identifiability problems, i.e., the inability to uniquely identify the unknown model parameters, making parameter estimation an even harder optimization task.

7 Experimental Evaluation

In this chapter, we evaluate ProBMoT, our system for inductive process modeling, by applying it to several real-world problems of automated modeling of aquatic ecosystems. Each of these problems has been previously addressed by applying LAGRAMGE 2.0 to measured data about the ecosystem at hand and the domain knowledge library for modeling aquatic ecosystems constructed by Atanasova et al. (2006b).

The four aquatic ecosystems considered are the lakes of Bled, Glumsø and Kasumigaura and the lagoon of Venice. The library of domain knowledge by Atanasova et al. (2006b) was adapted to use the ProBMoT formalism, described in Chapters 3 and 4. The process-based modeling formalisms of ProBMoT and LAGRAMGE 2.0 are quite similar, which facilitated the adaptation of the library.

Originally, (slightly) different versions of the library were used for each ecosystem. Slightly different IPM tasks were addressed, involving different kinds of nutrients, phytoplankton and zooplankton, as well as different conceptual processes. Here, all of these are put side-by-side, making very clear the similarities and differences between the different datasets and modeling tasks.

A major difference between the ProBMoT and LAGRAMGE 2.0 platforms lies in the parameter optimization approaches employed. LAGRAMGE 2.0 relied on the use of a standard-issue derivative-based local-optimization approach (ALG-717), combined with random restarts to ease the problem of multiple local optima. ProBMoT, on the other hand, also includes global meta-heuristic optimization approaches, based on ant-colony optimization (DASA) and evolutionary optimization (DE).

Our experimental evaluation focuses on investigating the effect that the parameter optimization approach used within ProBMoT has on the automated modeling process overall. In particular, we compare ProBMoT using ALG-717, which is very close to LAGRAMGE 2.0, and ProBMoT using DASA, a recently developed well-performing global meta-heuristic optimization approach. We are especially interested in the interplay between parameter optimization and model structure selection, since the error of a model (obtained after fitting its parameters) is used to perform the latter.

The evaluation is performed in the context of descriptive (as opposed to predictive) modeling, where the model is established to explain the observed behavior (training data) and not to predict future system behavior (test data). This is in line with previous inductive process modeling approaches and their evaluation. Therefore, we do not apply the usual train/test split of the data sets, nor do we apply cross-validation procedures. In consequence, we do not evaluate whether ProBMoT and its parameter estimation methods overfit the training data—since we are addressing an explanatory modeling task. However, we are aware that the issue of overfitting, especially in the context of better parameter estimation procedures, is very relevant. Evaluating ProBMoT in the context of establishing predictive models of dynamical systems is discussed as part of the further work in Chapter 8.

7.1 Experimental Setup

The aim of this study is to compare the influence of two established methods for parameter estimation, ALG-717 and DASA, on the overall process of automated modeling of aquatic ecosystems. The study addresses the task of modeling phytoplankton dynamics in four different aquatic ecosystems. First, we developed a library for modeling aquatic ecosystems, which we used as domain knowledge. Second, we obtained data from four ecosystems, relevant for modeling phytoplankton dynamics. Finally, we formulated conceptual models for each ecosystem. The resulting experimental setup consisted of 21 tasks of modeling phytoplankton dynamics, where a task is defined by the combination of a conceptual model and a data set. Each task was given to ProBMoT using both ALG-717 and DASA as parameter estimation methods, yielding a total of 42 experiments. The details of the experimental setup are given in the following subsections.

7.1.1 A Library for Modeling Aquatic Ecosystems

Using the formalism for representing domain knowledge for building process-based models presented in Chapter 3, we developed a library for modeling aquatic ecosystems. The library is based on the work for process-based modeling by (Atanasova et al., 2006b), where a similar library was developed using a different formalism. The entire library is given in Table B.1 in Appendix A.

The generalized conceptual model for modeling aquatic ecosystems underlying the library is given in Figure 7.1. The rounded rectangles represent the entity types and include *Nutrient*, *Primary Producer* and *Zooplankton.* The boxes represent the interactions between the entities in the form of process types. The library is constructed around the primary producer as a central entity. It contains processes suitable for modeling the dynamics of the primary producer. The main processes that affect the dynamics are growth, respiration, mortality, sedimentation and grazing. In the following subsections, we briefly present the modeling knowledge encoded in the library.

The Growth of Primary Producers

The growth process positively influences the concentration of the primary producer and can be stated as:

$$\frac{d}{dt} pp.conc = pp.growthRate \times pp.conc \tag{7.1}$$

where *pp.conc* is the concentration of the primary producer and *pp.growthRate* is the primary producer growth rate. The growth rate itself can be formulated as a limited growth rate, influenced by temperature, light and nutrient limitation functions:

$$pp.growthRate = pp.maxGrowthRate \times pp.tempGrowthLim \times pp.lightLim \times pp.nutrientLim \tag{7.2}$$

where *pp.maxGrowthRate* is the maximal growth rate under optimal conditions, *pp.tempGrowthLim* is the temperature influence on the growth rate, *pp.lightLim* is the influence of light on the growth rate, and *pp.nutrientLim* is the product of the limitation functions of all nutrients that are relevant for the growth of the primary producer.

Each type of limitation can be modeled with several different limitation functions that are present in the library. The influence of temperature on the growth can be modeled with two linear functions and an exponential function. In addition to this, temperature limitation can be turned off, by setting it to 1, which yields 4 alternatives for modeling the temperature influence. Similarly, light influence can be turned off or modeled with Monod or the optimal light limitation function, whereas nutrient limitation (for each nutrient) can be turned off or modeled with one of the Monod, Monod2 and exponential functions.

If all limitations are turned off (equal to 1), then the *pp.growthRate* coefficient is equal to *pp.maxGrowthRate*, i.e., is constant and the growth process is formulated as non limited, i.e. exponential growth function.

Respiration

The respiration of a primary producer decreases its mass, i.e., the phytoplankton mass. It can be expressed as an exponential decay or can be temperature-influenced. In the case where respiration is temperature-influenced, the influence of temperature can be expressed in terms analog to the influence of temperature on phytoplankton growth, i.e., no limitation at all or limitation in the form of one of two linear functions and one exponential limitation function.

Mortality

The mortality process represents non-predatory mortality and is included in scenarios where there is no grazing process included. The mortality can be expressed as an exponential decay of phytoplankton (first order kinetics) or second order kinetics. The process may be temperature-limited. The temperature limitation functions in the library include two linear functions and one exponential function.

Grazing

The grazing process included in the library is formulated for zooplankton filter-feeders as:

$$\frac{d}{dt}pp.conc = -zoo.maxFiltrationRate \times zoo.tempGrowthLim \times zoo.phytoLim \times zoo.conc \times pp.conc \tag{7.3}$$

where *zoo.maxFiltrationRate* is the maximal filtration rate coefficient, *zoo.conc* is the zooplankton concentration and *pp.conc* is the concentration of the phytoplankton. The temperature influence on grazing is specified through the *zoo.tempGrowthLim* term, which contains a linear or an exponential temperature limitation function option.

Sedimentation

Loss of phytoplankton biomass due to sedimentation is formulated by using the sedimentation rate coefficient, depth of the water column and the present concentration of the phytoplankton. The process may be temperature-influenced, where the temperature functions include two linear functions and one exponential function.

7.1.2 Data Sets

The data used for this study comes from four different ecosystems, namely Lake Bled in Slovenia, Lake Glumsø in Denmark, Lake Kasumigaura in Japan, and the lagoon of Venice in Italy. For each aquatic ecosystem, a number of physical, chemical and biological parameters were measured regularly for a sustained period of time in order to obtain the resulting data sets. Table 7.1 provides a summary of the data sets.

Lake Bled is a typical subalpine lake of glacial-tectonic origin. It occupies an area of 1.4 km^2 with a maximum depth of 30.1 m and an average depth of 17.9 m. The data set about the lake (obtained from the Slovenian Environmental Agency) comprises measurements of physical, chemical and biological parameters from 1995 to 2002 with a monthly frequency. The data used for modeling are as follows: temperature, light, dissolved inorganic nutrients

Table 7.1: Summary of measured data for the modeled ecosystems.

	Bled	Glumsø	Kasumigaura	Venice
Environmental influence	Temperature Light	Temperature Light	Temperature Light	Temperature
Nutrients	Phosphorus Nitrogen Silica	Phosphorus Nitrogen	Phosphorus Nitrogen	Phosphorus Nitrogen Ammonia
Primary producer	Phytoplankton	Phytoplankton	Phytoplankton (as Chl-a)	Algal biomass of *Ulva Rigida*
Zooplankton	Zooplankton (*Daphnia Hyalina*)	Zooplankton	None	None
Years	1995-2002	1973/74 1974/75	1986-1992	Loc 0: 1985/86 Loc 1,2,3: 1990/91
Number of data sets	8	2	7	4

in the lake (phosphorus, nitrogen and silica), and total phytoplankton biomass, and the zooplankton species *Daphnia hyalina*. The data were used as daily interpolated values between the measured points, obtained by using cubic spline interpolation (Atanasova et al., 2006c).

Lake Glumsø is situated in a sub-glacial valley in Denmark. It is a shallow lake with an average depth of about 2 m and a surface area of 266,000 m^2. The data set for Lake Glumsø includes daily measurements of water temperature, inorganic soluble nitrogen, soluble phosphorous, total phytoplankton, and zooplankton. In this case, we used two years of daily measurements from April 1973 to April 1974 and from October 1974 to October, 1975 (Atanasova et al., 2008).

Lake Kasumigaura is a shallow lake in Japan with an average depth of 4 m. It has a volume of 662 million m^3 and a surface area of 220 km^2. The lake's dataset comprises measurements from 1986 to 1992 of: water temperature, global radiation, dissolved inorganic phosphorus, total phytoplankton, measured as chlorophyll-a (chl-a). The measurements were used as interpolated values between the actual measured values using linear interpolation. The actual frequency of the measurements is monthly (Atanasova et al., 2006a).

The Lagoon of Venice has a surface area of 550 km^2, with an average depth of less than 1 m. The data set used here includes weekly measurements for slightly more than one year at four different locations (0, 1, 2, and 3) in the Lagoon. Location 0 was sampled in 1985/86, locations 1, 2, and 3 in 1990/91. The sampled quantities are nitrogen in ammonia (nh), nitrogen in nitrate (no), phosphorus in orthophosphate, temperature, and algal biomass (*Ulva rigida*). Related modeling experiments with this data set were described by (Atanasova, 2006).

We assemble the data sets according to domains and years. This yields eight data sets from Lake Bled, two from Glumsø, seven from Kasumigaura and four from Venice, giving a total of 21 data sets.

7.1.3 Inductive Process Modeling Tasks

In this study, we focus on the task of modeling phytoplankton dynamics using data from the domains presented in the previous section. For each domain (ecosystem), we prepare a conceptual model appropriate for modeling that particular case. The conceptual model is crafted in the formalism presented in Chapter 3. It includes the entities for which we have

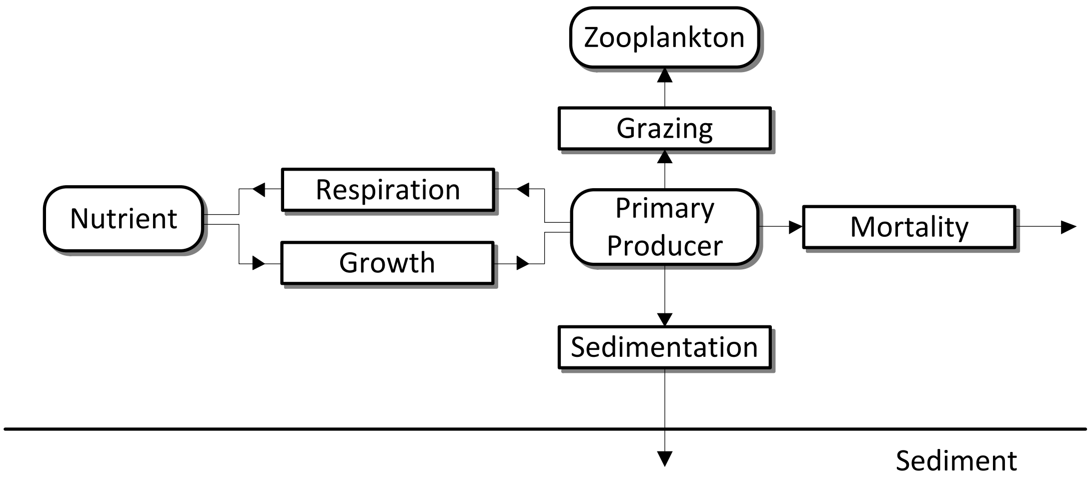


Figure 7.1: The generalized scheme of processes for modeling conceptual models, underlying the library of domain knowledge in aquatic ecosystems.

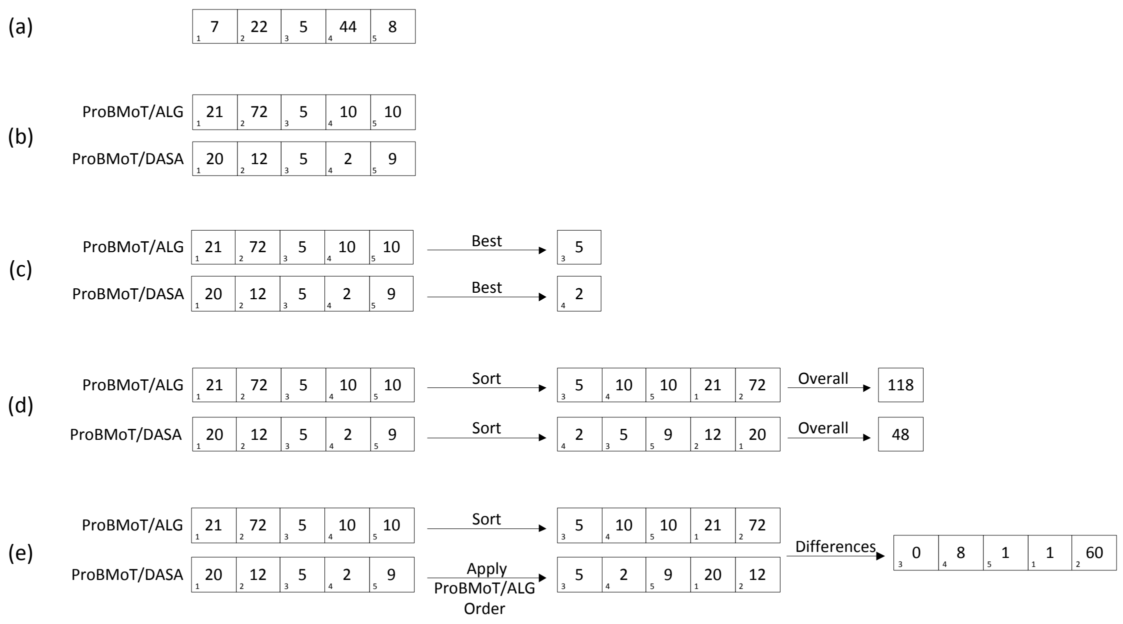


Figure 7.2: Schematic representation of the evaluation methodology. (a) Vector of model errors; (b) ProBMoT/ALG and ProBMoT/DASA vectors of model errors; (c) Best Model Improvement; (d) Overall Model Improvement; (e) Individual Model Improvement.

Table 7.2: A summary of the modeling tasks per domain (ecosystem).

	Bled	Glumsø	Kasumigaura	Venice
Entities	Phosphorus Nitrogen Silica Phytoplankton Zooplankton	Phosphorus Nitrogen Phytoplankton Zooplankton	Phosphorus Nitrogen Phytoplankton Zooplankton	Phosphorus Nitrate Ammonia Phytoplankton
Conceptual processes	Growth Respiration Grazing Sedimentation	Growth Respiration Grazing Sedimentation	Growth Respiration Mortality Sedimentation	Growth Respiration Mortality Sedimentation
Candidate models	27216	3024	5832	18144

measurements and the conceptual top-level processes appropriate for modeling phytoplankton dynamics: growth, respiration, mortality, sedimentation, and grazing by zooplankton. For two of the domains, Bled and Glumsø, there is data about zooplankton concentration. In these cases, we include a grazing process in the conceptual models. For Kasumigaura and Venice, where there is no data about zooplankton species, we include a natural mortality process instead. The conceptual models are outlined in Table 7.2.

For each domain, we tailor the general library presented in Section 7.1.1. to the requirements of the particular modeling task. We include processes which are needed for completing the conceptual model of the system. Based on previous analyses and previous knowledge about the domain, specific process alternatives which are appropriate for the given domain are taken into account, whereas unsuitable alternatives are discarded. Information about which processes are included in the pertinent libraries is given in Table 7.3.

For each modeling task, starting with a conceptual model and a library of domain knowledge, ProBMoT generated all specific candidate models structures. For the Bled domain it generated 27216 candidate models. The Glumsø task yielded 3024 candidates, the Venice task 5832 and Kasumigaura task 18144 candidate models. The constant parameters of each generated candidate model structure are fitted with ALG-717 and DASA.

The data from the domains are divided into separate data sets for each year. In the case of Venice, where data were collected at different locations, one data set corresponds to one location where the measurements were taken. In our experiments, the goal is to construct a separate model for each data set. The model is an explanatory model of the system for the given time range.

7.2 Evaluation Methodology

Each run of ProBMoT generates a series of N candidate model structures and applies parameter estimation method to obtain the models m_i, $i = 1, \ldots, N$. For each candidate model structure ProBMoT calculates its error $RMSE(m_i)$ on the provided data set. The resulting vector of model errors is schematically presented in Figure 7.2(a), where the number in the lower left corner of each box represents the sequential number of the model structure (as generated by ProBMoT) and the number in the center of the box represents the model error (for the set of parameter values estimated by ProBMoT). Let us denote the models obtained by ProBMoT/ALG and ProBMoT/DASA with m_i^A and m_i^D, respectively. To evaluate the impact of the optimization method on the ProBMoT performance, we thus compare the vectors $(RMSE(m_1^A), RMSE(m_2^A), \ldots, RMSE(m_N^A))$ and $(RMSE(m_1^D), RMSE(m_2^D), \ldots, RMSE(m_N^D))$ (Figure 7.2(b)). We emphasize here three aspects of this comparison, depicted in Figure

Table 7.3: Outline of the process templates modeled in the specific libraries for each modeling task.

Process template	Bled	Glumsø	Kasumigaura	Venice
NutrientPrimaryProducerInteraction	✓	✓	✓	✓
LightInfluence	✓	✓	✓	✓
NoLightLim				✓
LightLim	✓	✓	✓	
MonodLightLim	✓	✓	✓	
OptimalLightLim	✓	✓	✓	
NutrientInfluence	✓	✓	✓	✓
NoNutrientLim				
NutrientLim	✓	✓	✓	✓
MonodNutrientLim	✓	✓	✓	✓
Monod2NutrientLim	✓	✓	✓	✓
ExponentialNutrientLim	✓	✓	✓	✓
Growth	✓	✓	✓	✓
GrowthRate	✓	✓	✓	✓
LimitedGrowthRate	✓	✓	✓	✓
TempGrowthInfluence	✓	✓	✓	✓
NoTempGrowthLim				
TempGrowthLim	✓	✓	✓	✓
Linear1TempGrowthLim	✓	✓	✓	✓
Linear2TempGrowthLim	✓	✓	✓	✓
ExponentialTempGrowthLim	✓	✓	✓	✓
RespirationPP	✓	✓	✓	✓
ExponentialRespirationPP	✓	✓	✓	
TempRespirationPP	✓	✓	✓	✓
Temp1RespirationPP	✓	✓	✓	✓
Temp2RespirationPP	✓	✓	✓	✓
TempRespInfluence	✓	✓	✓	✓
NoTempRespLim				
TempRespLim	✓	✓	✓	✓
Linear1TempRespLim	✓	✓	✓	✓
Linear2TempRespLim	✓	✓	✓	✓
ExponentialTempRespLim	✓	✓	✓	✓
MortalityPP			✓	✓
ExponentialMortalityPP				✓
TempMortalityPP			✓	✓
Temp2MortalityPP			✓	✓
TempMortInfluence			✓	✓
NoTempMortLim				✓
TempMortLim			✓	✓
Linear1TempMortLim			✓	✓
Linear2TempMortLim			✓	✓
ExponentialTempMortLim			✓	✓
Sedimentation	✓	✓	✓	✓
TempSedInfluence	✓	✓	✓	✓
NoTempSedLim	✓	✓	✓	✓
TempSedLim	✓	✓	✓	
Linear1TempSedLim	✓	✓	✓	
Linear2TempSedLim	✓	✓	✓	
ExponentialTempSedLim	✓	✓	✓	
FeedsOn	✓	✓		
FeedsOnFiltration	✓	✓		
PhytoLim	✓	✓		
NoPhytoLim				
MonodPhytoLim	✓	✓		
Monod2PhytoLim	✓	✓		
ExponentialPhytoLim				

7.2(c, d & e), that correspond to the three central questions below.

Best Model Improvement What is the difference between the errors of the best models obtained with ProBMoT/DASA and ProBMoT/ALG?

Most importantly, we want to compare the best model found by ProBMoT/DASA to the best model found by ProBMoT/ALG as depicted in Fig 7.2(c). This is the key aspect of the comparison, since the modeler focuses his attention to the best model found by ProBMoT. Our hypothesis is that ProBMoT/DASA leads to the best model with lower error. Note that the best models found by ProBMoT/DASA and ProBMoT/ALG can differ not only in the parameter values but also it the model structure.

Overall Model Improvement What is the overall difference between the errors of the models obtained with ProBMoT/DASA and ProBMoT/ALG?

Comparing the best models is an important aspect of comparison, but only provides a part of the whole picture. Here, we extend our attention from the best models to overall model comparison as depicted in Figure 7.2(d). We first rank the models obtained by ProBMoT/ALG and ProBMoT/DASA with respect to their errors, i.e., we obtain two ranked lists of models, such that $RMSE(m^A_{a_1}) \leq RMSE(m^A_{a_2}) \leq \ldots \leq RMSE(m^A_{a_N})$ and $RMSE(m^D_{d_1}) \leq RMSE(m^D_{d_2}) \leq \ldots \leq RMSE(m^D_{d_N})$. We proceed with comparison of these two ranked lists as follows.

First, we check the extent to which ProBMoT/DASA outperforms ProBMoT/ALG, i.e., whether only the best ProBMoT/DASA model outperforms the best ProBMoT/ALG model or this holds for a wider range of models. We will be able to identify the ranges of models where ProBMoT/DASA is better and possibly where ProBMoT/ALG is better. Our hypothesis is that ProBMoT/DASA will outperform ProBMoT/ALG on the whole range of models.

Second, it will show whether the amount by which ProBMoT/DASA outperforms ProBMoT/ALG on the i$^{\text{th}}$ best model, as measured by the difference $RMSE(m^A_{a_i}) - RMSE(m^D_{d_i})$, increases or decreases as we move towards lower ranked models. Clearly, lower ranked models will have larger model errors, but we are also interested in the way the model error increases. Slow increase in model error means that we have a number of candidate models which have virtually equal performance and choosing among them is subject to other non-trivial constraints. Very rapid increase in the model error makes the distinction between good and bad models clear and makes choosing a final model an easier task. It is preferable to have a large error increase at the beginning of the ranked list so one can clearly distinguish between a few good and a majority of bad models. Our hypothesis is that ProBMoT/DASA provides a more clear distinction between good and bad models than ProBMoT/ALG.

Finally, we measure of the overall performance of ProBMoT/ALG and ProBMoT/DASA, by summing up the errors of all candidate models:

$$Overall_A = \sum_{i=1}^{N} RMSE(m^A_i) \qquad Overall_D = \sum_{i=1}^{N} RMSE(m^D_i) \tag{7.4}$$

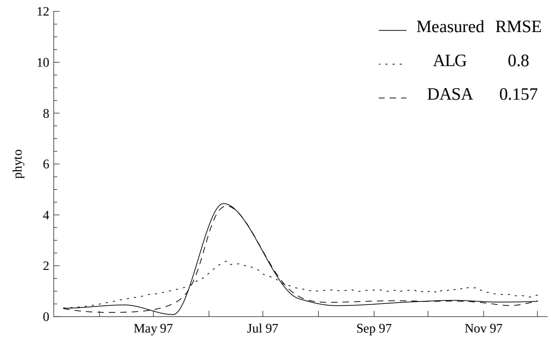
Individual Model Improvement What is the difference between the errors of each candidate model structure obtained with ProBMoT/DASA and ProBMoT/ALG?

In the previous type of comparison, the models are ordered according to ascending $RMSE$. When we are comparing the i$^{\text{th}}$ best ProBMoT/ALG model $m_{a_i}(p^A_{a_i})$ to the i$^{\text{th}}$ best ProBMoT/DASA model $m_{d_i}(p^D_{d_i})$, we are in general comparing two different model structures, i.e., $a_i \neq d_i$ in the general case. In the final and most stringent comparison, we compare the performance of ProBMoT/DASA and ProBMoT/ALG on the same model structure, i.e., $m_i(p^D_i)$ and $m_i(p^A_i)$, for $i = 1, \ldots, N$.

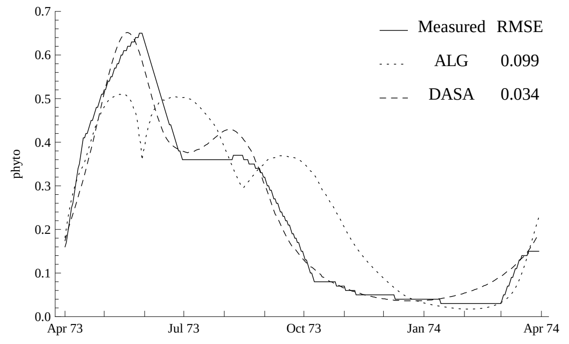
Table 7.4: Root mean squared error (RMSE) of the best models found with ProBMoT/ALG and ProBMoT/DASA.

	Bled								Glumsø	
	'95	'96	'97	'98	'99	'00	'01	'02	'73	'74
ProBMoT/ALG	0.650	0.362	0.800	0.802	1.282	2.439	0.683	0.771	0.099	0.074
ProBMoT/DASA	0.376	0.206	0.157	0.443	1.205	0.821	0.455	0.240	0.034	0.030

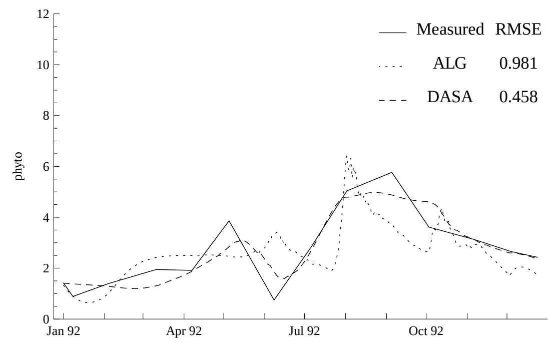
	Kasumigaura							Venice			
	'86	'87	'88	'89	'90	'91	'92	L0	L1	L2	L3
ProBMoT/ALG	1.685	0.775	0.674	0.782	0.648	0.819	0.981	138.417	221.619	111.924	55.987
ProBMoT/DASA	1.249	0.646	0.381	0.417	0.350	0.766	0.458	83.431	203.007	88.459	43.085



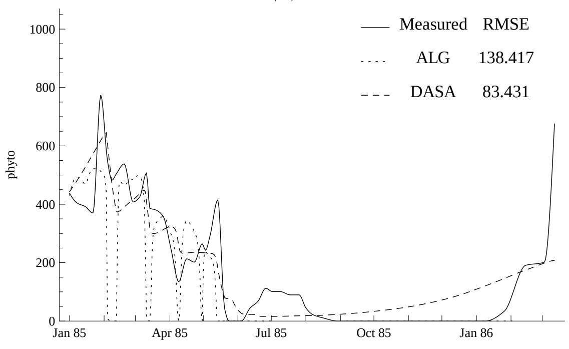
(a)



(b)



(c)



(d)

Figure 7.3: Simulations of the best models found with ProBMoT/ALG and ProBMoT/DASA on several data sets. (a) Bled '97; (b) Glumsø '73; (c) Kasumigaura '92; (d) Venice 0.

To perform this comparison, we use the ranking of models obtained by ALG to rank both list of models, i.e.: $RMSE(m^A_{a_1}) \leq RMSE(m^A_{a_2}) \leq \ldots \leq RMSE(m^A_{a_N})$ and $RMSE(m^D_{a_1}), RMSE(m^D_{a_2}), \ldots, RMSE(m^D_{a_N})$ as depicted in Figure 7.2(e). We then observe the distribution of differences $RMSE(m^A_{a_i}) - RMSE(m^D_{a_i})$ for $i = 1, \ldots, n$, where the value of n ranges from 2 through 10, 100, 1000, (and 10000, if $N > 10000$) to N.

7.3 Results and Discussion

In this section, we present and discuss the results of the empirical evaluation. The analysis is based on the comparison of the performance of ProBMoT when using the ALG-717 (ProBMoT/ALG) and the DASA (ProBMoT/DASA) optimization algorithms for solving the parameter estimation task.

Most of the computation time (over 99%) of ProBMoT is spent on numerical simulation. Therefore, the number of evaluations of the objective function is the key parameter which is controlled during optimization. Both DASA and ALG-717 are given the same number of evaluations. The number of evaluations during the optimization of each candidate model

Table 7.5: Total root mean squared error (Total-RMSE) of all models generated with ProBMoT/ALG and ProBMoT/DASA.

	Bled ($\times 10^6$)								Glumsø ($\times 10^3$)	
	‘95	‘96	‘97	‘98	‘99	‘00	‘01	‘02	‘73	‘74
ProBMoT/ALG	23.44	22.48	14.59	27.01	107.6	46.64	39.32	29.89	39.01	38.07
ProBMoT/DASA	3.802	7.442	3.247	7.058	9.555	7.802	15.05	1.915	18.73	13.79

	Kasumigaura ($\times 10^6$)							Venice ($\times 10^9$)			
	‘86	‘87	‘88	‘89	‘90	‘91	‘92	L0	L1	L2	L3
ProBMoT/ALG	128.1	42.52	36.74	30.75	39.6	30.26	48.84	111.1	152.0	108.4	9.234
ProBMoT/DASA	58.54	5.836	11.78	17.86	7.295	13.58	10.47	87.09	138.2	73.17	7.608

is computed as 5000 times the number of unknown parameters of the model. DASA is an iterative method and the number of evaluations can be directly set as a parameter. ALG-717, on the other hand, is a deterministic method and converges within certain number of steps (evaluations) depending on the complexity of the landscape. ProBMoT restarts ALG-717 a certain number of times in order to avoid being trapped in a local optimum. The number of restarts ProBMoT performs is set so that the total number of evaluations of ALG-717 matches that of DASA, i.e., is equal to 5000 times the number of unknown parameters.

7.3.1 Best Model Improvement

Table 7.4 shows the ***RMSE*** values for the best models found with ProBMoT/ALG and ProBMoT/DASA on all data sets. Since the models selected by ProBMoT/DASA always have a lower ***RMSE*** than those selected by ProBMoT/ALG, we can conclude that DASA outperforms ALG on each modeling task.

Figure 7.3 shows the simulations of the best ProBMoT/ALG and the best ProBMoT/-DASA model on some of the tasks, along with the measured data. There is a considerable difference in the errors of the best models found on the Bled ’97 task. This quantitative difference in the ***RMSE*** values is directly visible as a qualitative difference in the simulations. The ALG model simulation exhibits some resemblance to the measured data, whereas the DASA model simulation follows the dynamics of the phytoplankton concentration very closely, both in terms of time of rapid phytoplankton growth and peak amplitude.

The Glumsø ’73 task gives similar results. The DASA model represents a close match of the original data, whereas the ALG model only gives a rough approximation, completely missing the main peak of phytoplankton concentration.

The Kasumigaura ’92 task is a more challenging task. On this task, ALG does not manage to find an admissible model which can be seen from the large error and the simulation, which follows the measured data very poorly. DASA does not excel on this task either, but still manages to roughly identify the two peaks in the phytoplankton concentration and their magnitude.

An even poorer performance of ALG is observed on the Venice data set at location 0. The model found by ALG produces erratic behavior and fails to capture any underlying dynamics. The DASA model represents a considerable improvement over ALG, despite of its poor overall quality. The simulations of the best models for all tasks are given in Appendix B. Our hypothesis that the best ProBMoT/DASA model outperforms the best ProBMoT/ALG model on each task is thus fully confirmed.

7.3.2 Overall Model Improvement

Figure 7.4 shows the error profiles for both ALG and DASA on the tasks from Figure 7.3. The profile curves on these tasks, as well as all the other tasks (presented in Appendix C.) clearly show that DASA manages to find better models throughout the space of model

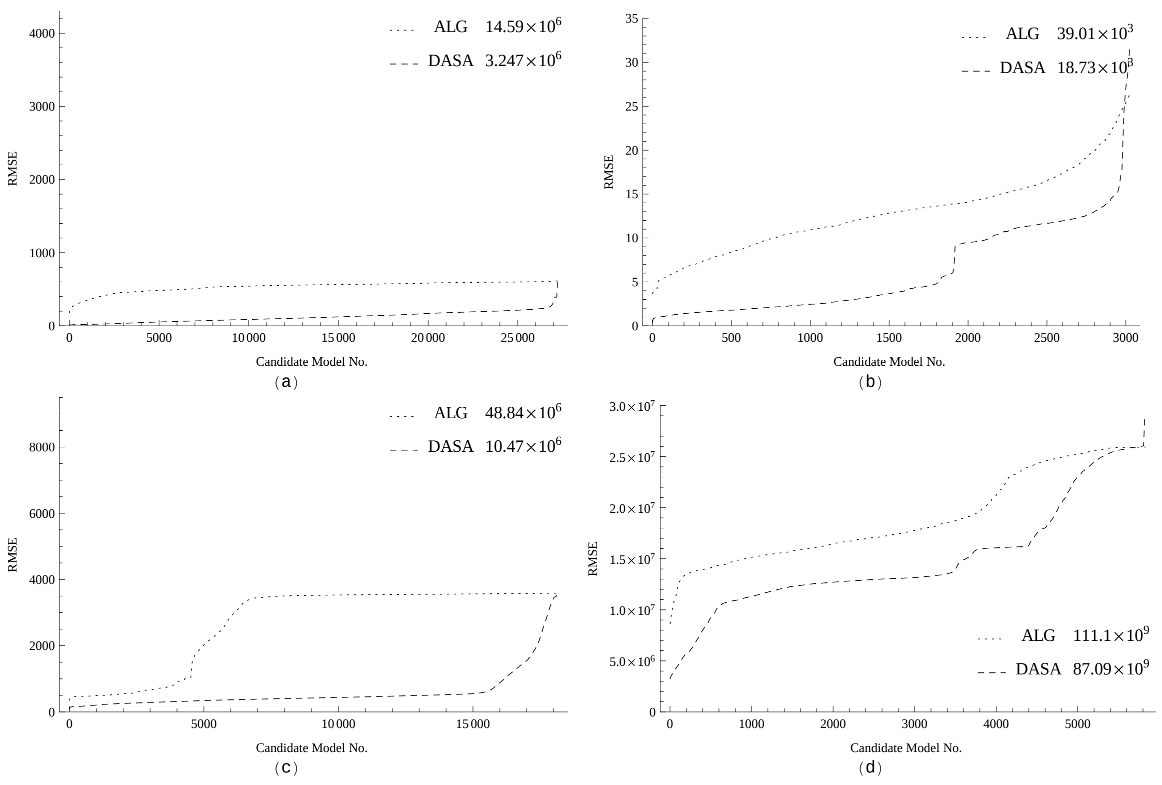


Figure 7.4: Error profile curves of models found with ProBMoT/ALG and ProBMoT/DASA on several data sets. (a) Bled '97; (b) Glumsø '73; (c) Kasumigaura '92; (d) Venice 0.

structures. In other words, not only the best model found with ProBMoT/DASA is better than the best one found with ProBMoT/ALG, but also the second best model found with ProBMoT/DASA is better than the second best model found with ProBMoT/ALG, the third best model from ProBMoT/DASA is better than the third best model from ProBMoT/ALG, and so forth. The only exceptions are the few models at the very tail of the ranked list of models. In these cases, both ALG and DASA find poor parameter values, but in some cases the models with parameter values found with DASA have larger *RMSE* than those with parameters found with ALG. Our hypothesis that ProBMoT/DASA outperforms ProBMoT/ALG, not only on the best models for each task, but on the complete range of models (with the exception of the very few models at the end) is thus confirmed.

Table 7.5presents the overall error of ProBMoT/ALG and ProBMoT/DASA on each task. It is clear that the total error of ProBMoT/DASA is smaller than the total error of ProBMoT/ALG on each task.

A closer inspection of the error profiles shows that the shapes of the profiles of ProBMoT/ALG and ProBMoT/DASA are very different. In general, the *RMSE* of the ProBMoT/ALG models increases rapidly with the increase in the rank number at the beggining of the error profile and remains steady from there on. Thus, ProBMoT/ALG clearly distinguishes a small number of good models from a large number of bad models. The opposite is true for the shape of the ProBMoT/DASA profile. A very large part of the models that appear at the beginning of the profile have indistinguishably similar *RMSE* values and a much smaller part of the models at the end have much larger *RMSE* values. Thus, DASA doesn not provide a useful distinction of good and bad models. This is contrary to our initial expectations, and our hypothesis that DASA will provide a more clear distinction of good and bad models is rejected. Within the conclusion and further work, we examine the possible reasons for this outcome and formulate a new hypothesis related to this behavior of ProBMoT/DASA.

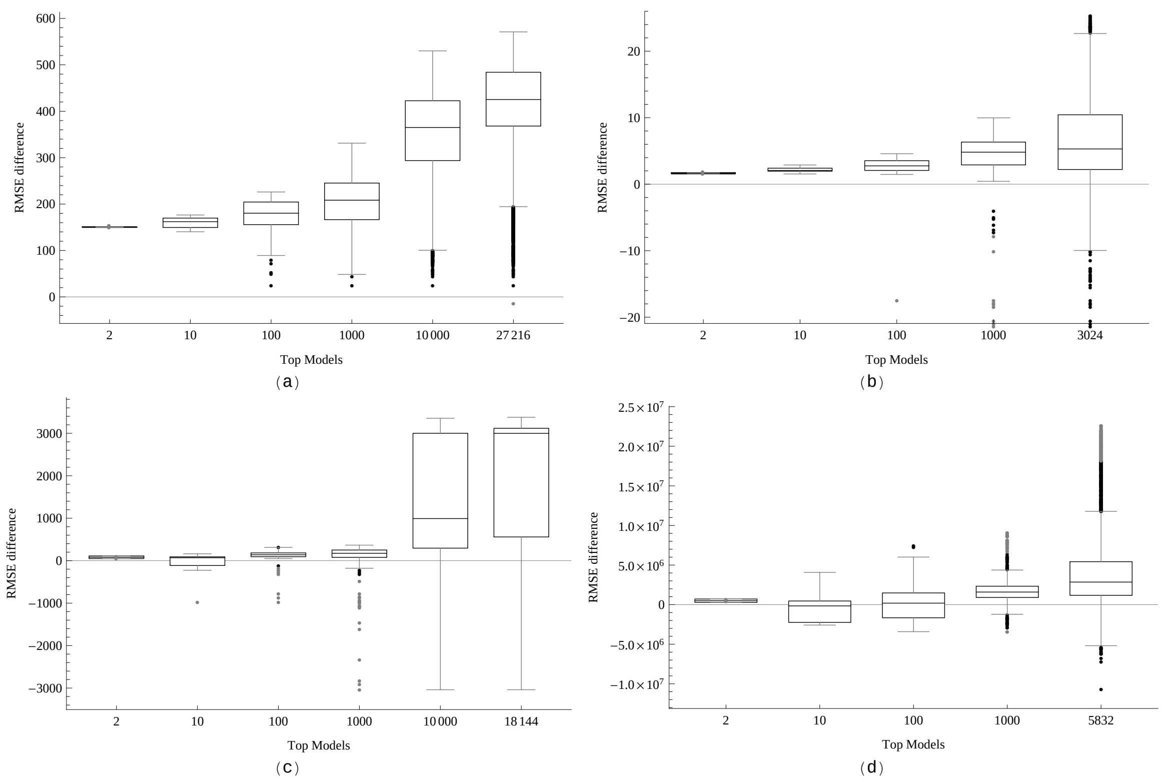


Figure 7.5: Distributions of the model-wise differences of errors between ProBMoT/ALG and ProBMoT/DASA. (a) Bled '97; (b) Glumsø '73; (c) Kasumigaura '92; (d) Venice 0.

7.3.3 Individual Model Improvement

Figure 7.5 shows the distributions of error differences between models fitted with DASA and with ALG for the tasks from Figure 7.3. The first box-plot in each figure represents a summary of the differences of the two best models found with ALG. If these differences are above zero, DASA does not miss the two best solutions found by ALG. Next, the distributions of differences for the 10, 100, 1000 and 10 000 (if applicable) best models are shown. Lastly, the distribution of differences for all models is shown. In all cases, there is a clear shift of the distribution away from zero, which indicates that DASA overall finds better parameter values than ALG for the same model structures. Moreover, in the vast majority of cases, DASA manages to find better parameter values for the model structures for which ALG finds the best values. Notable exceptions are Kasumigaura '86 and Venice 3 where DASA does not manage to find better parameter values than ALG on several best structures identified by ALG. The distributions of error differences for all tasks are given in Appendix D.

7.4 Conclusion

In this chapter, we performed an extensive experimental evaluation of ProBMoT. Data from four different aquatic ecosystems were used, collected over different lengths of time, ranging from two to eight years. A library of domain knowledge for modeling this type of ecosystems was used as well. For the four different ecosystems and different periods of time, we addressed 21 different automated modeling tasks.

The focus of the experimental evaluation was the effect of the use of different methods for parameter estimation within ProBMoT. Unlike previous approaches to inductive process modeling, ProBMoT can use both local and global optimization approaches. We investigated the effects of substituting DASA, a global search method for the previously used local search method ALG-717.

The results conclusively show that DASA outperforms ALG-717 on all modeling tasks. Not only ProBMoT using DASA manages to find better models for the systems, DASA manages to find better parameter values across the whole spectrum of model structures. Furthermore, refitting the best model structure found by ALG with DASA yields better results in almost all cases.

Automated modeling has so far focused mostly on discovering single year models of aquatic ecosystems and this is the approach taken in the present experimental evaluation. The reason for this is that experiments with discovering models which hold for longer periods of time yielded poor results in earlier work. We conjecture that this is largely due to the poor parameter estimation when using local search methods. Using global search methods opens the opportunity to tackle long term modeling of dynamical systems with automated modeling tools.

One clue to support this conjecture lies in the error profile curves. Almost without exception, ALG manages to find good parameter values for very few model structures which can be seen by the shape of the error profile curve which increases rapidly at the beginning and reaches a plateau of models with high error values afterwards. The error profile curves of DASA instead show a plateau of models with low error values which are equally good, and poor parameter values for very few model structures which can be seen by the increase in error values near the end of the curve.

This means that the small data sets used do not provide enough information to discriminate among the different model structures. It is evident that it is not easy to select an appropriate model structure, or more precisely, that a single one-year data batch does not provide enough information to narrow down the choice to a single (or a few) model structure(s). It is thus necessary to use more data on several years (a longer batch or several one-year batches) to obtain additional information that would further narrow down the choice among the large number of model structures. Hence, we need to use longer, multi-year data sets, to narrow down the choice of model structures.

8 Conclusion

In this thesis, we presented a formalism for modeling dynamical systems. The formalism describes a dynamical system as a process-based model, consisting of entities and processes. Entities and processes can optionally be structured within compartments.

The process-based formalism enables us to express background knowledge in the same language as the model. The knowledge about entities, processes, and compartments is abstracted into templates, which are the main components of knowledge about a given domain. These templates are gathered into a library of domain-specific background knowledge, which formalizes the knowledge we have about a given domain.

We presented ProBMoT, a platform for automated modeling of dynamical systems that takes into account knowledge expressed in the process-based formalism. Using an incompletely specified conceptual model of the system, the library of domain-specific knowledge, and measurement data for a particular system, it identifies both the structure and numerical parameters of a model for the system.

We applied ProBMoT on a case study of modeling phytoplankton dynamics in four aquatic ecosystems. We adapted an existing domain-specific library for modeling aquatic ecosystems, which we used as background knowledge. The experiments compared the influence of two established methods for parameter estimation, ALG-717 and DASA, on the overall process of automated modeling of aquatic ecosystems.

In the remainder of this chapter, we summarize the main contributions of the thesis and outline some directions for further work.

8.1 Summary of Contributions

The contributions of the dissertation are along three main lines, summarized in the following subsections:

8.1.1 Process-Based Formalism

The thesis presented a novel formalism for representing process-based models and background knowledge. This process-based formalism integrates several improvements over previous formalisms. In this way, it directly addresses their limitations.

We developed a mathematical notation to express the relations that hold between the components of the process-based formalism. The mathematical notation encompasses the components of process-based models, as well as background knowledge. The structure of models and libraries is represented through mathematical functions and relations. The semantics of the process-based libraries and models are expressed as logical statements that must hold. This notation, presented in Chapter 4, is then utilized to define incompletely specified models in Chapter 5, describe the algorithms of the ProBMoT platform in Chapter 6, and present the results of the empirical case study in Chapter 7.

We augmented process-based models and background knowledge with compartments, which are used to structure the content of process-based models. With their use, we can represent real-world dynamical systems as hierarchical multi-compartment models. To the best of our knowledge, this is the first process-based formalism (inductive process modeling approach) that allows for (the learning of) compartmental models.

We represent a conceptual model of a system with an incompletely specified process-based model—a process-based model which is incomplete in the structural specification or in the numerical parameters. Incomplete models, described in detail in Chapter 5, serve as a means to specify the partial knowledge one has about the system that is being modeled, thus defining the task and space of feasible model candidates for the ProBMoT platform. While previous IPM approaches have used implicit definitions of incomplete models (such as task specifications), they have not formalized and defined them clearly, precisely and explicitly.

8.1.2 Inductive Process Modeling Platform ProBMoT

We developed the IPM platform ProBMoT, which implements the proposed process-based formalism for representing models, incomplete models, and libraries. As presented in Chapter 6, ProBMoT implements the complete pipeline of inductive process modeling including enumeration of all candidate model structures and parameter estimation. ProBMoT includes several improvements that give it a competitive advantage over existing IPM methods.

ProBMoT supports arbitrary observational scenarios. Observational scenarios are defined by an explicit output specification which includes not only observable and hidden variables, but arbitrary algebraic equations that can involve all model variables and constants. The parameters of these algebraic equations can be fitted as well.

Besides the local method used in earlier IPM approaches, ProBMoT also includes global optimization methods for the parameter estimation stage. The global methods are based on meta-heuristic optimization approaches, such as differential evolution and ant-colony optimization. The global optimization methods outperform local optimization methods in finding parameter values for the model structures considered by ProBMoT.

ProBMoT allows the use of various quality criteria of model fitness. ProBMoT integrates different quality criteria in the form of various error function. These error functions can serve as objective functions for the optimization algorithms for parameter estimation.

8.1.3 Empirical Evaluation of ProBMoT

We applied ProBMoT to modeling phytoplankton growth in four different aquatic ecosystems. These are the lakes of Bled, Glumsø and Kasumigaura, and the lagoon of Venice. The case study, presented in Chapter 7, yielded several important results.

The case study demonstrated the suitability of the ProBMoT platform for automated modeling of highly nonlinear dynamical systems, such as aquatic ecosystems. Describing an aquatic ecosystem with a process-based model was intuitive and appealing to the domain experts. Defining the phenomena in the system as processes quantified with equations very closely resembles the natural way ecological modelers approach the modeling process. Moreover, this case study illustrated the appropriateness of capturing domain-specific knowledge with libraries of entity and process templates. The case study was focused on building explanatory models of the aquatic ecosystems. Hence, the performance of the models was evaluated on the training sets.

We identify three criteria for comparing the performance of parameter estimation methods in the context of structure identification. Best model improvement is the simplest criterion which compares only the best models found by ProBMoT with the different parameter estimation techniques. Overall model improvement compares all models found using the different parameter estimation techniques, paired according to rank. Finally, individual model improvement pairs models with the same model structure and assesses the relative performance of the parameter estimation techniques for each model structure.

The case study reiterates a statement known from numerous previous studies,i.e., that global optimization methods perform better than local optimization methods at finding suitable values for parameters of ODEs. We extend those claims to the task of automated modeling and demonstrate that metaheuristics outperform local search when dealing with multiple candidate model structures. The use of better parameter optimization methods highlights and emphasizes the need for larger amounts of high-quality data to distinguish between alternative model structures.

8.2 Further Work

We present four main directions for further (and, to a certain extent, ongoing) work.

8.2.1 More Extensive Experimental Evaluation

The first direction for further work concerns additional experimental evaluation of ProBMoT. The work presented in this dissertation focuses on the development of the ProBMoT platform, the process-based formalism it uses, and the underlying algorithms. The experimental evaluation constitutes a more modest part of the presented work.

Further systematic experimental evaluation is required to reaffirm the conclusions drawn in this dissertation. The further experimentation should be carried out along three main branches.

The first one is an experimental study based on synthetic data. Starting with a known model, the model is simulated and the resulting simulation is used as input to the algorithm. The goal in such an exercise is to reconstruct the behavior of the model taken as ground truth.

The second branch is experimental evaluation of ProBMoT on the task of predictive modeling. Our work, up until this point, has mainly focused on building explanatory models. When building predictive models, the issue of overfitting the available data should be properly addressed. Therefore, it is essential to introduce new objective functions based on the MDL principle (Rissanen, 1978), which will take into account the complexity of the model structure, i.e., introduce preference-bias towards simpler model structures.

The third branch of experimental evaluation should focus on evaluating ProBMoT in different scientific domains. The principal domain of focus in this dissertation has been that of aquatic ecosystems. Modeling aquatic ecosystems is a complex task and the sufficient modeling knowledge that exists in this domain makes it a very good candidate for inductive process modeling.

Another domain in which modeling knowledge becomes increasingly abundant and better organized is systems biology. First steps in formulating a library of background knowledge suitable for modeling biochemical cellular processes have already been made. It is expected that further development of such knowledge and experimental results will follow in due course.

8.2.2 From Short-Term to Long-Term Models

Automated modeling of aquatic ecosystems with IPM approaches has so far focused mostly on discovering single-year models and this is the approach we have taken in our work. The reason for this is that experiments with discovering models which hold for longer periods of time yielded poor results in earlier work. We conjecture that this is largely due to the poor parameter estimation when using local search methods. Using global search methods opens the opportunity to tackle long term modeling of dynamical systems with automated modeling tools.

One clue to support this lies in the error profile curves for model structures considered by ProBMoT, produced with different parameter estimation techniques (Chapter 7). Almost without exception, local methods manage to find good parameter values for very few model structures which can be seen by the shape of the error profile curve which increases rapidly at the beginning and reaches a plateau of models with high error values afterwards. The error profile curves of global methods instead show a plateau of models with low error values which are equally good, and poor parameter values for very few model structures which can be seen by the increase in error values near the end of the curve. This means that the small data sets used do not provide enough information to discriminate among the different model structures. Hence, we need to use longer, multi-year data sets, to narrow down the choice of model structures.

8.2.3 Heuristic Exploration of Candidate Model Structures

The space of all possible candidate model structures that is defined by an incompletely specified model and a library is very complex in nature. The work presented in this thesis approaches the removal of incompleteness in stages, thus creating a sequential model generation procedure. In this procedure, all completions of the incomplete model structure are considered. In practice, a very large number of such structures may result, thus making the approach computationally prohibitive.

Alternative direction that should be explored is implementing a heuristic search procedure that will replace the exhaustive enumeration of candidate model structures. Searching the space of model structures heuristically has the advantage of traversing only a subset of the search space and is more time-efficient. It can thus be used to tackle much larger problems for which enumerating all structures would not be feasible.

8.2.4 Further Improvements of the Parameter Estimation Subsystem

Most of the work presented in the dissertation concentrates on the expressiveness of the process-based formalism and the generation of candidate model structures. In practice, the usefulness of IPM methods is largely limited by the efficiency of method for parameter estimation that is used. We do focus on development of methods for parameter estimation, and instead integrate well-established, proven nonlinear optimization methods to solve the parameter estimation task. ProBMoT's architecture is flexible and allows effortless integration of additional optimization methods.

An important direction of further work is the integration of additional methods for parameter estimation, especially global methods for nonlinear optimization. The global methods integrated in ProBMoT have performed well at the BBOB 2009 benchmark[1], which has been superseded by the BBOB 2013 benchmark. We intent to integrate some of the methods that have performed best at the BBOB 2013, and in particular CMA-ES (Hansen and Ostermeier, 1996) which has shown excellent results.

[1]http://coco.gforge.inria.fr

9 Acknowledgments

First and foremost, I would like to acknowledge the role of my supervisor Saso Džeroski and co-supervisor Ljupčo Todorovski in the process of creating this dissertation. From the very beginning, when it was but a drawing board sketch, until the very last version of the manuscript, their ideas, guidance, and support were an immense contribution to this dissertation. For this, they will have my long-lasting gratitude and appreciation.

Many thanks to the members of the committee that evaluated my dissertation: Jurij Šilc, Juš Kocijan, and Michele Sebag. I am grateful for their incredibly fast and precise evaluation and for providing constructive comments that undoubtedly improved the quality of the manuscript.

I would like to thank my co-authors Saso Džeroski, Ljupčo Todorovski, Katerina Taškova, Nataša Atanasova, Mateja Škerjanec, and Boris Kompare for the fruitful collaboration that resulted in the publications that were the foundation for this dissertation.

I would like to acknowledge the financial support of the Slovenian Research Agency through the young researcher program. In addition, I acknowledge the funding that I received through the EU funded project SUMO—Supermodeling by combining imperfect models (7.FP STREP 288708, ICT FET), on which I collaborated.

I wish to thank all members of the Department of Knowledge Technologies at the Jožef Stefan Institute. They provided the pleasant working environment without which it would not have been possible to persist in my research pursuits. I would especially like to thank all of my officemates, past and present, Marko Debeljak, Valentin Gjorgjioski, Aneta Trajanov, Darko Aleksovski, Jovan Tanevski, Dragana Miljković, and Matic Perovšek for making my time at the office more enjoyable.

Finally, my biggest gratitude goes to my parents, Svetlana and Riste, and to my brother Davor for their encouragement, understanding, and patience on the long and challenging road that brought me here. Without your love and support, none of this would have been possible.

10 References

Abadi, M. and Cardelli, L. (1998). *A Theory of Objects*. Springer.

Aoki, N. and Hiraide, K. (1994). *Topological Theory of Dynamical Systems: Recent Advances*, volume 52 of *North-Holland Mathematical Library*. Elsevier.

Ascher, U. M. and Petzold, L. R. (1998). *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics.

Atanasova, N. (2006). *Preparation and use of the domain expert knowledge for automated modeling of aquatic ecosystems*. PhD thesis, Faculty of Civil and Geodetic Engineering, University of Ljubljana.

Atanasova, N., Recknagel, F., Todorovski, L., Džeroski, S., and Kompare, B. (2006a). Computational assemblage of ordinary differential equations for chlorophyll-a using a lake process equation library and measured data of Lake Kasumigaura. In Recknagel, F., editor, *Ecological Informatics*, pages 409–427. Springer Berlin Heidelberg.

Atanasova, N., Todorovski, L., Džeroski, S., and Kompare, B. (2006b). Constructing a library of domain knowledge for automated modelling of aquatic ecosystems. *Ecological Modelling*, 194(1–3):14–36.

Atanasova, N., Todorovski, L., Džeroski, S., and Kompare, B. (2008). Application of automated model discovery from data and expert knowledge to a real-world domain: Lake Glumso. *Ecological Modelling*, 212(1-2):92–98.

Atanasova, N., Todorovski, L., Džeroski, S., Rekar Remec, v., Recknagel, F., and Kompare, B. (2006c). Automated modelling of a food web in lake Bled using measured data and a library of domain knowledge. *Ecological Modelling*, 194(1–3):37–48.

Atkinson, K. (1989). *An Introduction to Numerical Analysis*. Wiley.

Box, G. E. P. and Draper, N. R. (1987). *Empirical model-building and response surface*. John Wiley & Sons, Inc., New York, NY, USA.

Bradley, E., Easley, M., and Stolle, R. (2001). Reasoning about nonlinear system identification. *Artificial Intelligence*, 133(1–2):139–188.

Braun, M. (1993). *Differential Equations and Their Applications*, volume 11 of *Texts in Applied Mathematics*. Springer.

Bredeweg, B., Linnebank, F., Bouwer, A., and Liem, J. (2009). Garp3 — workbench for qualitative modelling and simulation. *Ecological Informatics*, 4(5–6):263–281.

Bridewell, W., Langley, P., Todorovski, L., and Džeroski, S. (2008). Inductive process modeling. *Machine Learning*, 71(1):1–32.

Brown, P. N., Byrne, G. D., and Hindmarsh, A. C. (1989). VODE: a variable-coefficient ODE solver. *SIAM Journal on Scientific and Statistical Computing*, 10(5):1038–1051.

Bunch, D. S., Gay, D. M., and Welsch, R. E. (1993). Algorithm 717: Subroutines for maximum likelihood and quasi-likelihood estimation of parameters in nonlinear regression models. *ACM Transactions on Mathematical Software*, 19(1):109–130.

Butcher, J. C. (2003). *Numerical Methods for Ordinary Differential Equations*. Wiley.

Cohen, S. D. and Hindmarsh, A. C. (1996). CVODE, a stiff/nonstiff ODE solver in C. *Computers in Physics*, 10(2):138–143.

Crawford, J., Farquhar, A., and Kuipers, B. (1990). Qpc: a compiler from physical models into qualitative differential equations. In *Proceedings of the eighth National conference on Artificial intelligence - Volume 1*, AAAI'90, pages 365–372. AAAI Press.

Dorigo, M. and Stützle, T. (2004). *Ant Colony Optimization*. MIT Press.

Džeroski, S., Goethals, B., and Panov, P., editors (2010). *Inductive Databases and Constraint-Based Data Mining*. Springer New York.

Džeroski, S. and Todorovski, L. (1993). Discovering dynamics. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 97–103. Morgan Kaufmann.

Džeroski, S. and Todorovski, L. (1995). Discovering dynamics: From inductive logic programming to machine discovery. *Journal of Intelligent Information Systems*, 4(1):89–108.

Džeroski, S. and Todorovski, L. (2001). Encoding and using domain knowledge on population dynamics for equation discovery. In Magnani, L., Nersessian, N. J., and Pizzi, C., editors, *Model-Based Reasoning: Scientific Discovery, Technological Innovation, Values (MBR'01)*, Pavia, Italy.

Džeroski, S. and Todorovski, L. (2002). *Logical and Computational Aspects of Model-Based Reasoning*, chapter Encoding and using domain knowledge on population dynamics for equation discovery, pages 227–247. Springer.

Džeroski, S. and Todorovski, L. (2003). Learning population dynamics models from data and domain knowledge. *Ecological Modelling*, 170:129–140.

Eiben, A. and Smith, J. (2003). *Introduction to Evolutionary Computing*. Springer.

Falkenhainer, B. and Michalski, R. (1986). Integrating quantitative and qualitative discovery: The abacus system. *Machine Learning*, 1(4):367–401.

Forbus, K. D. (1984). Qualitative process theory. *Artificial Intelligence*, 24(1–3):85–168.

Gear, C. (1971). *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice Hall.

Gershenfeld, N. (1998). *The Nature of Mathematical Modeling*. Cambridge University Press.

Gordon, M. J. (1988). *Programming Language Theory and Its Implementation*. Prentice Hall.

Gray, G. J., Murray-Smith, D. J., Li, Y., Sharman, K. C., and Weinbrenner, T. (1998). Nonlinear model structure identification using genetic programming. *Control Engineering Practice*, 6(11):1341–1352.

Gunter, C. A. and Mitchell, J. C., editors (1994). *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* MIT Press.

Gupta, G. K., Sacks-Davis, R., and Tescher, P. E. (1985). A review of recent developments in solving odes. *ACM Computing Surveys*, 17(1):5–47.

Hansen, N., Auger, A., Ros, R., Finck, S., and Posik, P. (2010). Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009. In *Workshop Proceedings of the GECCO Genetic and Evolutionary Computation Conference.* ACM.

Hansen, N. and Ostermeier, A. (1996). Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 312–317.

Hindmarsh, A. C. (1980). LSODE and LSODI, two new initial value ordinary differnetial equation solvers. *ACM SIGNUM Newsletter*, 15(4):10–11.

Hindmarsh, A. C. (1983). ODEPACK, a systematized collection of ODE solvers. In Stepleman, R. S., editor, *Scientific Computing.* IMACS Transactions on Scientific Computation.

Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S. (2005). Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396.

Horst, R., Pardalos, P., and Thoai, N. (2000). *Introduction to global optimization.* Springer.

Iserles, A. (1996). *A First Course in the Numerical Analysis of Differential Equations.* Cambridge University Press.

Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562.

Kellert, S. (1993). *In the wake of chaos : unpredictable order in dynamical systems.* University of Chicago Press, Chicago.

Kitano, H. (2001). *Foundations of Systems Biology.* MIT Press.

Kokar, M. (1986). Determining arguments of invariant functional descriptions. *Machine Learning*, 1(4):403–422.

Korošec, P., Šilc, J., and Filipič, B. (2012). The differential ant-stigmergy algorithm. *Information Sciences*, 192:82–97.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.

Križman, V., Džeroski, S., and Kompare, B. (1995). Discovering dynamics from measured data. *Electrotechnical Review*, 62(3–4):191–198.

Kuipers, B. (1994). *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge.* MIT Press, Cambridge, MA.

Langley, P., George, D., Bay, S., and Saito, K. (2003). Robust induction of process models from time-series data. In *Proceedings of the Twentieth International Conference on Machine Learning*, ICML 2003, pages 432–439.

Langley, P., Sánchez, J., Todorovski, L., and Džeroski, S. (2002). Inducing process models from continuous data. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pages 347–354, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Langley, P., Simon, H. A., Bradshaw, G., and Żytkow, J. (1987). *Scientific Discovery: Computational Explorations of the Creative Processes.* MIT Press.

Laniak, G. F., Olchin, G., Goodall, J., Voinov, A., Hill, M., Glynn, P., Whelan, G., Geller, G., Quinn, N., Blind, M., Peckham, S., Reaney, S., Gaber, N., Kennedy, R., and Hughes, A. (2013). Integrated environmental modeling: A vision and roadmap for the future. *Environmental Modelling & Software*, 39(0):3–23.

Ljung, L. (1999). System identification. In Webster, J. G., editor, *Wiley Encyclopedia of Electrical and Electronics Engineering*, volume 21, pages 263–282. John Wiley & Sons, New York.

Luenberger, D. G. (1979). *Introduction to Dynamic Systems: Theory, Models, and Applications.* Wiley.

Mitchell, J. C. (1996). *Foundations for Programming Languages.* MIT Press.

Mitchell, T. (1997). *Machine Learning.* McGraw-Hill.

Muggleton, S. (1991). Inductive logic programming. *New Generation Computing*, 8(4):295–318.

Muggleton, S. (1999). Inductive logic programming: Issues, results and the challenge of learning language in logic. *Artificial Intelligence*, 114(1–2):283–296.

Nelles, O. (2001). *Nonlinear System Identification: From Classical Approaches to Neural Networks and Fuzzy Models.* Springer.

Pfanzagl, J. (1994). *Parametric Statistical Theory.* Walter de Gruyter.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes. The Art of Scientific Computing.* Cambridge University Press, 3rd edition edition.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14(5):465–471.

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach.* Prentice Hall.

Saad, Y. and Schultz, M. H. (1986). GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869.

Sampine, L. and Gordon, M. (1975). *Computer Solution of Ordinary Differential Equations: the Initial Value Problem.* Freeman.

Schaffer, C. (1993). Bivariate scientific function finding in a sampled, real-data testbed. *Machine Learning*, 12(1–3):167–183.

Schmidt, M. and Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science*, 324:81–85.

Tashkova, K., Šilc, J., Atanasova, N., and Džeroski, S. (2012). Parameter estimation in a nonlinear dynamic model of an aquatic ecosystem with meta-heuristic optimization. *Ecological Modelling*, 226:36–61.

Todorovski, L. (2003). *Using domain knowledge for automated modeling of dynamic systems with equation discovery.* PhD thesis, University of Ljubljana.

Todorovski, L., Bridewell, W., Shiran, O., and Langley, P. (2005). Inducing hierarchical process models in dynamic domains. In *Proceedings of the 20th national conference on Artificial intelligence*, pages 892–897. AAAI Press.

Todorovski, L. and Džeroski, S. (1997). Declarative bias in equation discovery. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 376–384. Morgan Kaufmann.

Todorovski, L. and Džeroski, S. (2006). Integrating knowledge-driven and data-driven approaches to modeling. *Ecological Modelling*, 194(1–3):3–13.

Törn, A., Ali, M., and Viitanen, S. (1999). Stochastic global optimization: Problem classes and solution techniques. *Journal of Global Optimization*, 14:437–447.

Verner, J. (1978). Explicit Runge-Kutta Methods with Estimates of the Local Truncation Error. *SIAM Journal on Numerical Analysis*, 15(4):772–790.

Verner, J. (1979). Families of Imbedded Runge-Kutta Methods. *SIAM Journal on Numerical Analysis*, 16(5):857–875.

Washio, T. and Motoda, H. (1997). Discovering admissible models of complex systems based on scale-types and identity constraints. In *In Proceedings of IJCAI'97, Vol.2*, pages 810–817.

Zembowicz, R. and Żytkow, J. (1992). Discovery of equations: experimental evaluation of convergence. In *Proceedings of the tenth national conference on Artificial intelligence*, AAAI'92, pages 70–75. AAAI Press.

Index of Figures

Index of Tables

Index of Algorithms

Appendix

A List of Publications Related to the Dissertation

A.1 Original Scientific Articles

- Čerepnalkoski, D., Taškova, K., Todorovski, L., Atanasova, N., and Džeroski, S. (2012) The influence of parameter fitting methods on model structure selection in automated modeling of aquatic ecosystems. *Ecological Modelling*, 245:136–166.

A.2 Published Scientific Conference Contribution

- Škerjanec, M., Čerepnalkoski, D., Džeroski, S., Kompare, B., and Atanasova, N. (2013) Modelling dynamic systems using a hybrid approach. In *Machine Learning in Water Systems: Proceedings of the AISB Convention 2013*, pages 35–38, University of Exeter.

- Čerepnalkoski, D., Todorovski, L., Atanasova, N., and Džeroski, S. (2009) Incorporating qualitative equations in process-based models. In *QR 2009, 23rd International Workshop on Qualitative Reasoning*, pages 136–141, University of Ljubljana.

- Čerepnalkoski, D., Taškova, K., Todorovski, L., and Džeroski, S. (2008) Inducing process-based models of dynamic systems from multiple data sets. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, ECML/PKDD 2008: Proceedings of Induction of process models, IPM'08*, pages 5–12, University of Antwerp.

- Čerepnalkoski, D., Džeroski, S., Taškova, K., and Todorovski, L. (2007) Learning generic models of dynamic systems. In *Proceedings of the 10th International Multiconference Information Society 2007*, pages 186–189, Jozef Stefan Institute.

A.3 Published Scientific Conference Contribution Abstract

- Čerepnalkoski, D., Taškova, K., Todorovski, L., Atanasova, N., and Džeroski, S. (2011) Influence of parameter fitting methods on model structure selection in automated modeling of aquatic ecosystems. In *Ecological Hierarchy from the Genes to the Biosphere : Book of Abstracts*, page 49, ECEM 2011, Trento, Italy.

- Čerepnalkoski, D. and Džeroski, S. (2009) Automated modeling of dynamic systems in systems biology. In *From Molecules to Function : Course Book : Program and Abstracts*, page 104, 3rd FEBS Advanced Lecture Course on Systems Biology, Alpbach, Austria.

- Čerepnalkoski, D., Todorovski, L., and Džeroski, S. (2009) Process-based formalism for modelling dynamical systems. In *Abstracts and Programm*, page 104, Workshop Data 2 Dynamics, Freising, Germany.

B Aquatic Ecosystems Library

Table B.1: The complete library of the domain knowledge for modeling aquatic ecosystems used for ProBMoT.

```
// ENTITIES

template entity EcosystemEntity {
  vars :
    conc {aggregation:sum; unit:"kg/m^3"; range:<0,inf>};
}

template entity Population : EcosystemEntity {
  vars:
    tempGrowthLim{aggregation:product},
    tempRespLim{aggregation:product},
    tempMortLim{aggregation:product},
    tempExcLim{aggregation:product},
    tempSedLim{aggregation:product};
}

template entity PrimaryProducer : Population {
  vars:
    nutrientLim{aggregation:product},
    lightLim{aggregation:product},
    growthRate;
  consts:
    maxGrowthRate { range: <0.05,3>; unit:"1/(day)"};
}

template entity Zooplankton : Population {
  vars:
    phytoLim{aggregation:sum},
    phytoSum{aggregation:sum};
  consts:
    maxFiltrationRate { range: <0.01, 15>; unit:"m3/(mgZoo*day)"},
    assimilationCoeff { range: <0,inf>; unit:"mgZoo/(mgAlgae)"};
}

template entity Nutrient : EcosystemEntity {
  consts:
    halfSaturation {range: <0,15>; unit:"mg/l"},
    alpha {range: <0,inf>; unit:"mgAlgaeBiomass/mgZooBiomass"};
}

template entity Environment {
  vars:
    temperature,light,flow;
  consts:
    volume,depth,area;
}

// PROCESSES

template process NutrientPrimaryProducerInteraction
(pp : PrimaryProducer, ns : Nutrient<1, inf>, env : Environment ) {
  processes:
    LightInfluence(pp, env), NutrientInfluence(pp, <n:ns>), Growth(pp, ns, env),
    RespirationPP(pp, ns, env);
}

// Temperature Growth Influence

template process TempGrowthInfluence(pop : Population, env : Environment) {}

template process NoTempGrowthLim : TempGrowthInfluence {
  equations:
    pop.tempGrowthLim = 1;
}

template process TempGrowthLim : TempGrowthInfluence {
  consts:
    refTemp { range: <10, 22>},
    minTemp { range: <0, 6>},
    optTemp { range: <15, 25>};
}

template process Linear1TempGrowthLim : TempGrowthLim {
  equations:
    pop.tempGrowthLim = env.temperature/refTemp;
}
```

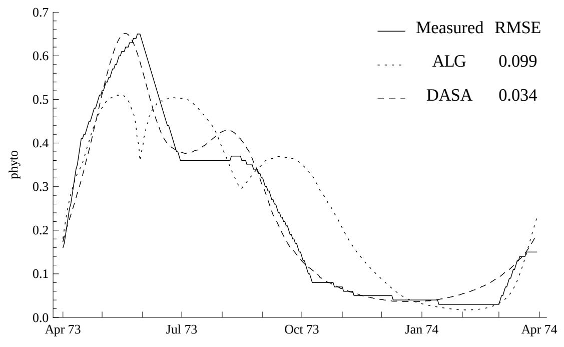
```
template process Linear2TempGrowthLim : TempGrowthLim {
  equations:
    pop.tempGrowthLim = (env.temperature - minTemp)/(refTemp - minTemp);
}

template process ExponentialTempGrowthLim : TempGrowthLim {
  consts:
    theta { range: <1.06, 1.13>};
  equations:
    pop.tempGrowthLim = pow(theta, env.temperature - refTemp);
}

// Temperature Respiration Influence

template process TempRespInfluence(pop : Population, env : Environment) {}

template process NoTempRespLim : TempRespInfluence {
  equations:
    pop.tempRespLim = 1;
}

template process TempRespLim : TempRespInfluence {
  consts:
    refTemp { range: <10, 22>},
    minTemp { range: <0, 6>},
    optTemp { range: <15, 25>};
}

template process Linear1TempRespLim : TempRespLim {
  equations:
    pop.tempRespLim = env.temperature/refTemp;
}

template process Linear2TempRespLim : TempRespLim {
  equations:
    pop.tempRespLim = (env.temperature - minTemp)/(refTemp - minTemp);
}

template process ExponentialTempRespLim : TempRespLim {
  consts:
    theta { range: <1.06, 1.13>};
  equations:
    pop.tempRespLim = pow(theta, env.temperature - refTemp);
}

// Temperature Mortality Influence

template process TempMortInfluence(pop : Population, env : Environment) {}

template process NoTempMortLim : TempMortInfluence {
  equations:
    pop.tempRespLim = 1;
}

template process TempMortLim : TempMortInfluence {
  consts:
    refTemp { range: <10, 22>},
    minTemp { range: <0, 6>},
    optTemp { range: <15, 25>};
}

template process Linear1TempMortLim : TempMortLim {
  equations:
    pop.tempRespLim = env.temperature/refTemp;
}

template process Linear2TempMortLim : TempMortLim {
  equations:
    pop.tempRespLim = (env.temperature - minTemp)/(refTemp - minTemp);
}

template process ExponentialTempMortLim : TempMortLim {
  consts:
    theta { range: <1.06, 1.13>};
  equations:
    pop.tempRespLim = pow(theta, env.temperature - refTemp);
}

// Temperature Sedimentation Influence

template process TempSedInfluence(pop : Population, env : Environment) {}
template process NoTempSedLim : TempSedInfluence {
  equations:
    pop.tempSedLim = 1;
}

template process TempSedLim : TempSedInfluence {
  consts:
    refTemp { range: <10, 22>},
    minTemp { range: <0, 6>},
    optTemp { range: <15, 25>};
}

template process Linear1TempSedLim : TempSedLim {
  equations:
    pop.tempSedLim = env.temperature/refTemp;
}

template process Linear2TempSedLim : TempSedLim {
  equations:
    pop.tempSedLim = (env.temperature - minTemp)/(refTemp - minTemp);
}

template process ExponentialTempSedLim : TempSedLim {
  consts:
    theta { range: <1.06, 1.13>};
  equations:
    pop.tempSedLim = pow(theta, env.temperature - refTemp);
}

// Light Influence

template process LightInfluence(pp: PrimaryProducer, env: Environment) {}

template process NoLightLim : LightInfluence {
  equations:
    pp.lightLim = 1;
}

template process LightLim : LightInfluence {}

template process MonodLightLim : LightLim {
  consts:
    halfSat {range: <0, 200>};
  equations:
    pp.lightLim = env.light / (env.light + halfSat);
}

template process OptimalLightLim : LightLim {
  consts:
    optLight {range: <100, 200>};
  equations:
    pp.lightLim = env.light * exp(- env.light / optLight + 1) / optLight;
}

// Nutrient Influence

template process NutrientInfluence(pp : PrimaryProducer, n : Nutrient) {}

template process NoNutrientLim : NutrientInfluence {
  equations:
    pp.nutrientLim = 1;
}

template process NutrientLim : NutrientInfluence {}

template process MonodNutrientLim : NutrientLim {
  equations:
    pp.nutrientLim = n.conc / (n.conc + n.halfSaturation);
}

template process Monod2NutrientLim : NutrientLim {
  equations:
    pp.nutrientLim = n.conc * n.conc / (n.conc * n.conc + n.halfSaturation);
}

template process ExponentialNutrientLim : NutrientLim {
  consts:
    saturationRate { range: <0, 15>};
  equations:
    pp.nutrientLim = 1 - exp(-saturationRate * n.conc);
}
```

```
// Growth

template process Growth(pp : PrimaryProducer, ns : Nutrient<1, inf>, env : Environment) {
  processes:
    TempGrowthInfluence(pp, env), GrowthRate(pp, ns, env);
  equations:
    td(pp.conc) = pp.growthRate * pp.conc,
    td(<n:ns>.conc) = -n.alpha * pp.growthRate * pp.conc;
}

template process GrowthRate(pp : PrimaryProducer, ns : Nutrient<1, inf>, env: Environment) {}

template process LimitedGrowthRate : GrowthRate {
  equations :
    pp.growthRate = pp.maxGrowthRate * pp.tempGrowthLim * pp.lightLim * pp.nutrientLim;
}

// Respiration PP

template process RespirationPP(pp : PrimaryProducer, ns : Nutrient<1, inf>, env: Environment) {}

template process ExponentialRespirationPP : RespirationPP {
  consts:
    respRate {range: <0.0001, 2>};
  equations:
    td(pp.conc) = -respRate * pp.conc,
    td(<n:ns>.conc) = respRate * pp.conc;
}

template process TempRespirationPP : RespirationPP {
  processes:
    TempRespInfluence(pp, env);
}

template process Temp1RespirationPP : TempRespirationPP {
  consts:
    respRate {range: <0.0001, 1>};
  equations:
    td(pp.conc) = -respRate * pp.conc * pp.tempRespLim,
    td(<n:ns>.conc) = respRate * pp.conc * pp.tempRespLim;
}

template process Temp2RespirationPP : TempRespirationPP {
  consts:
    respRate {range: <0.0001, 1>};
  equations:
    td(pp.conc) = -respRate * pp.conc * pp.conc * pp.tempRespLim,
    td(<n:ns>.conc) = respRate * pp.conc * pp.conc * pp.tempRespLim;

}

// Mortality PP

template process MortalityPP(pp : PrimaryProducer, env : Environment) {
  processes:
    TempMortInfluence(pp, env);
}

template process ExponentialMortalityPP : MortalityPP {
  consts:
    mortRate {range: <0.0001, 2>};
  equations:
    td(pp.conc) = -mortRate * pp.conc;
}

template process TempMortalityPP : MortalityPP {
  consts:
    mortRate {range: <0.0001, 2>};
  equations:
    td(pp.conc) = -mortRate * pp.conc * pp.tempMortLim;
}

template process Temp2MortalityPP : MortalityPP {
  consts:
    mortRate {range: <0.0001, 2>};
  equations:
    td(pp.conc) = -mortRate * pp.conc * pp.conc * pp.tempMortLim;
}

// Feeds On

template process FeedsOn(zoo:Zooplankton, pps:PrimaryProducer<1,inf>, env: Environment){
  processes :
    TempGrowthInfluence(zoo, env), PhytoLim(zoo, pps);
}
```

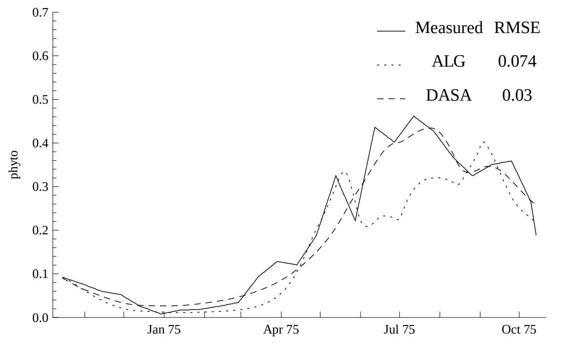
```
template process FeedsOnFiltration: FeedsOn{
  equations :
    td(zoo.conc) = zoo.assimilationCoeff * zoo.maxFiltrationRate * zoo.tempGrowthLim
    * zoo.conc * zoo.phytoSum * zoo.phytoLim,
    td(<pp:pps>.conc) = - zoo.maxFiltrationRate * zoo.tempGrowthLim * zoo.conc
    * pp.conc * zoo.phytoLim;
}

template process PhytoLim(zoo: Zooplankton, pps : PrimaryProducer<1,inf>) {}

template process NoPhytoLim : PhytoLim {
  equations:
    zoo.phytoLim = 1;
}

template process MonodPhytoLim : PhytoLim {
  consts:
    halfSaturation {range: <0, 20> };
  processes:
    Summation(zoo, pps);
  equations:
    zoo.phytoLim = zoo.phytoSum/ (halfSaturation+ zoo.phytoSum);
}

template process Monod2PhytoLim : PhytoLim {
  consts:
    halfSaturation {range: <0, 20> };
  processes:
    Summation(zoo, pps);
  equations:
    zoo.phytoLim = zoo.phytoSum * zoo.phytoSum / (zoo.phytoSum * zoo.phytoSum + halfSaturation);
}

template process ExponentialPhytoLim : PhytoLim {
  consts:
    saturationRate {range: <0, 5> };
  processes:
    Summation(zoo, pps);
  equations:
    zoo.phytoLim = 1 - exp(-saturationRate * zoo.phytoSum);
}

template process Summation(zoo : Zooplankton, pps: PrimaryProducer<1,inf>) {
  equations:
    zoo.phytoSum = <pp:pps>.conc;
}

template process Sedimentation(pop : Population, env: Environment) {
  processes:
    TempSedInfluence(pop, env);
  consts:
    sedimentationRate { range: <0.0001, 0.5>; unit:"1/(day)"};
  equations:
    td(pop.conc) = -(sedimentationRate / env.depth) * pop.conc * pop.tempSedLim;
}
```

C Detailed Experimental Results

C.1 Model Simulations

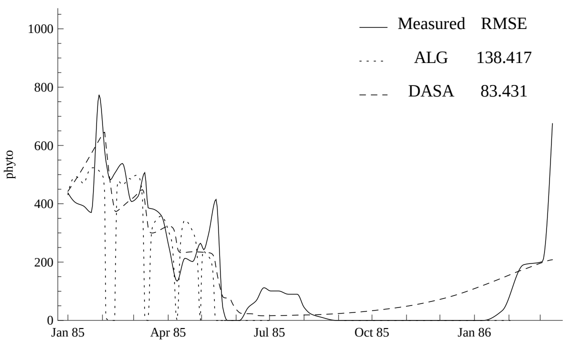


(a)

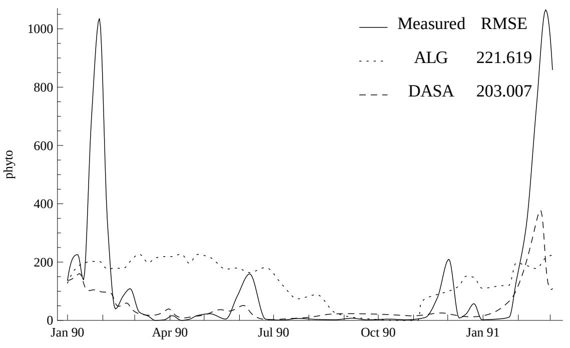


(b)

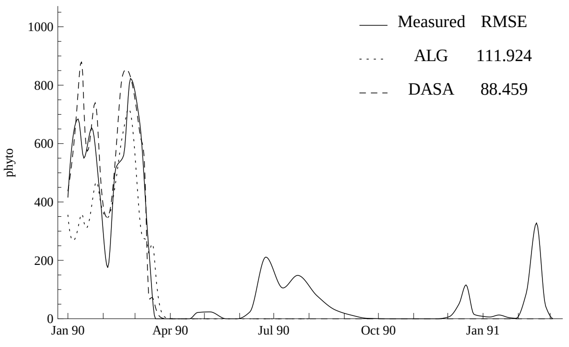
Figure C.1: Simulations of the best models found by ProBMoT/ALG and ProBMoT/DASA for each task for the Glumsø domain. (a) Glumsø '73; (b) Glumsø '74.



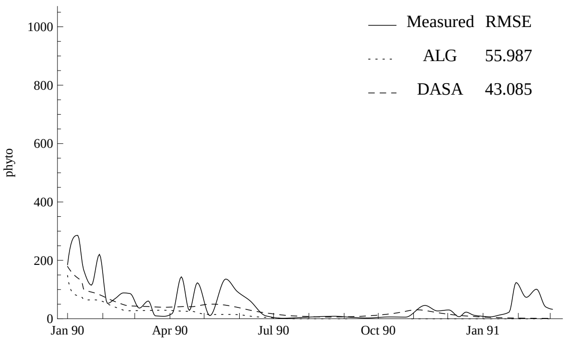
(a)



(b)

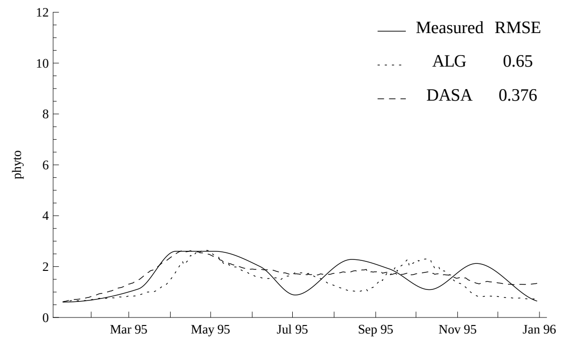


(c)

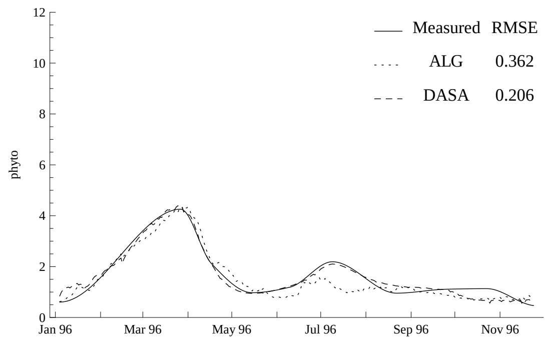


(d)

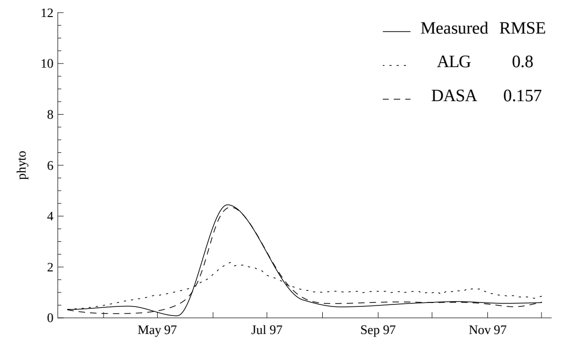
Figure C.2: Simulations of the best models found by ProBMoT/ALG and ProBMoT/DASA for each task for the Venice domain. (a) Venice Loc. 0; (b) Venice Loc. 1; (c) Venice Loc. 2; (d) Venice Loc. 3.



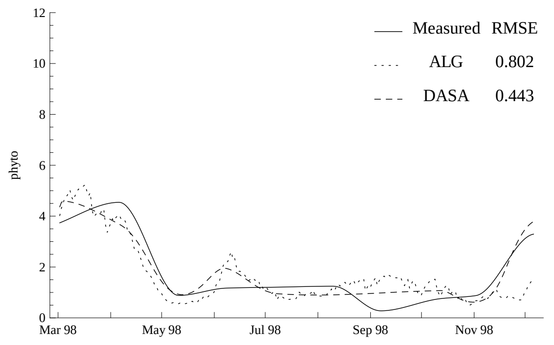
(a)



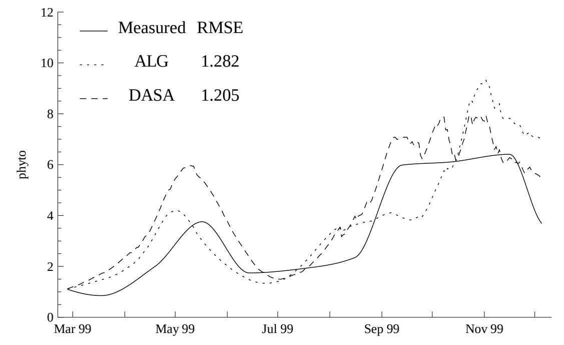
(b)



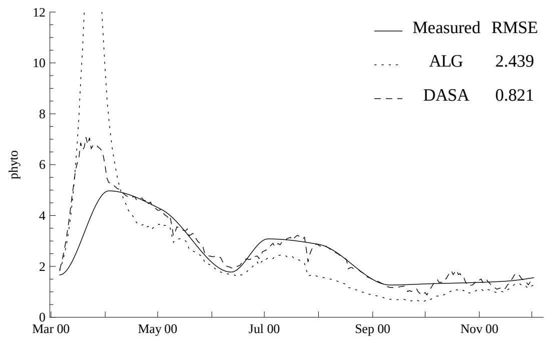
(c)



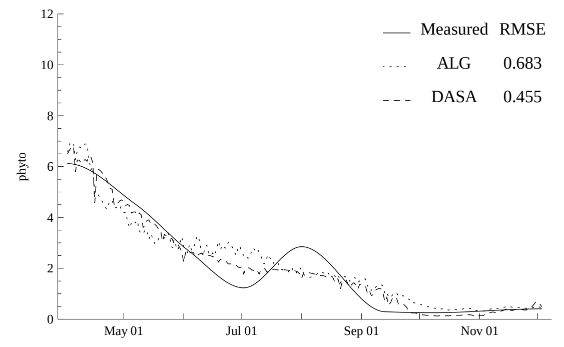
(d)



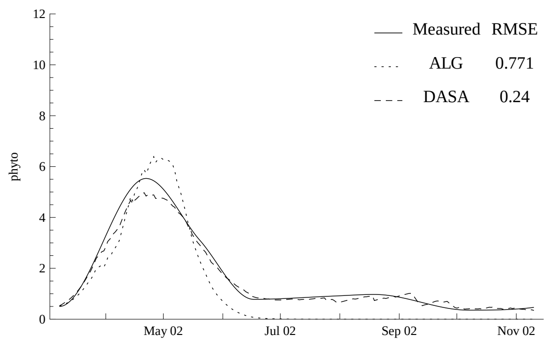
(e)



(f)

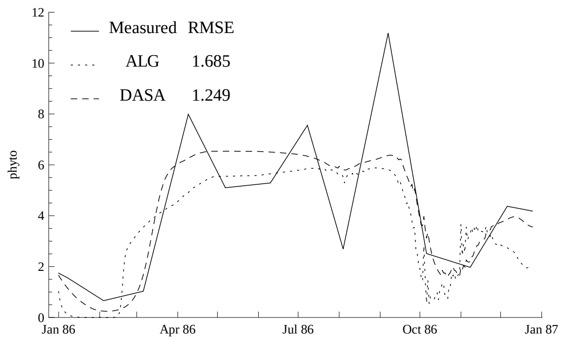


(g)

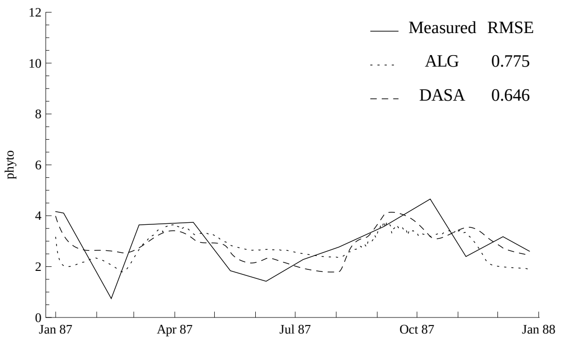


(h)

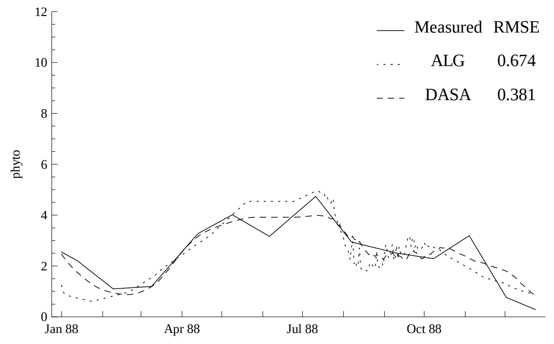
Figure C.3: Simulations of the best models found by ProBMoT/ALG and ProBMoT/DASA for each task for the Bled domain. (a) Bled '95; (b) Bled '96; (c) Bled '97; (d) Bled '98; (e) Bled '99; (f) Bled '00; (g) Bled '01; (h) Bled '02.



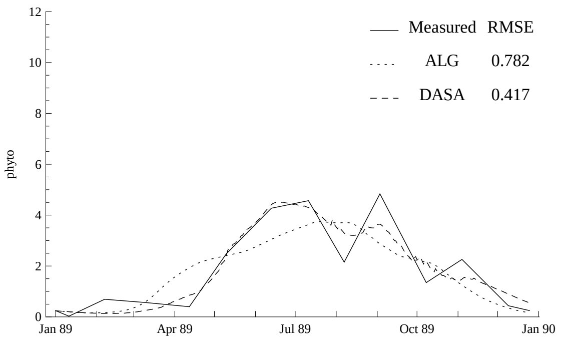
(a)



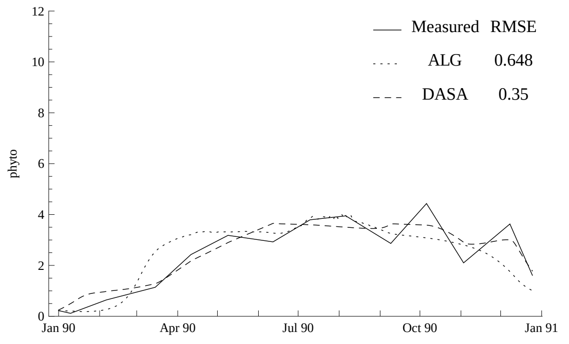
(b)



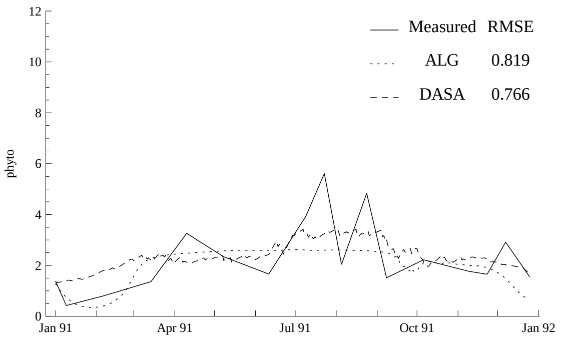
(c)



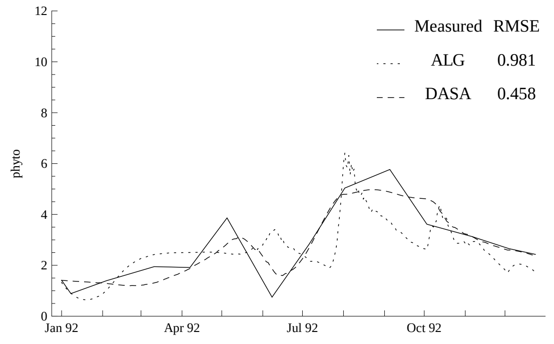
(d)



(e)



(f)



(g)

Figure C.4: Simulations of the best models found by ProBMoT/ALG and ProBMoT/DASA for each task for the Kasumigaura domain. (a) Kasumigaura '86; (b) Kasumigaura '87; (c) Kasumigaura '88; (d) Kasumigaura '89; (e) Kasumigaura '90; (f) Kasumigaura '91; (g) Kasumigaura '92.

C.2 Error Profiles

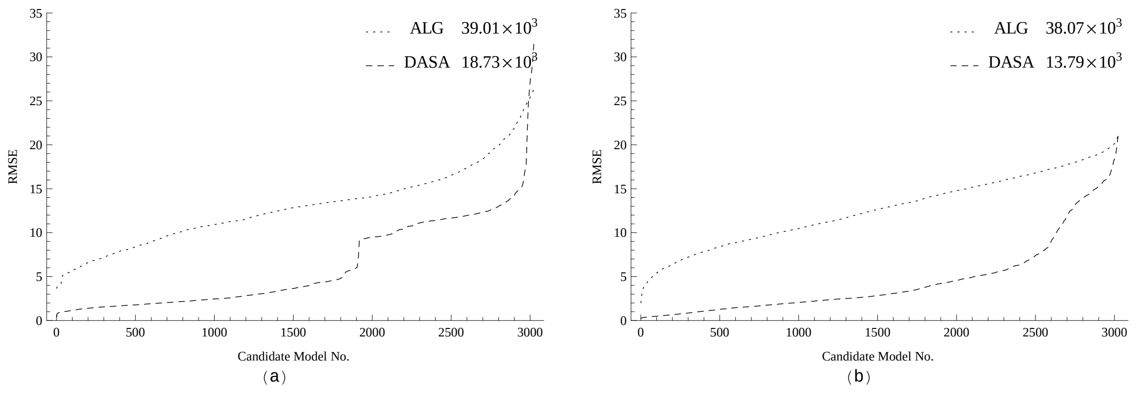


Figure C.5: Error profiles for ProBMoT/ALG and ProBMoT/DASA on all automated modeling tasks for the Glumsø domain. (a) Glumsø '73; (b) Glumsø '74.

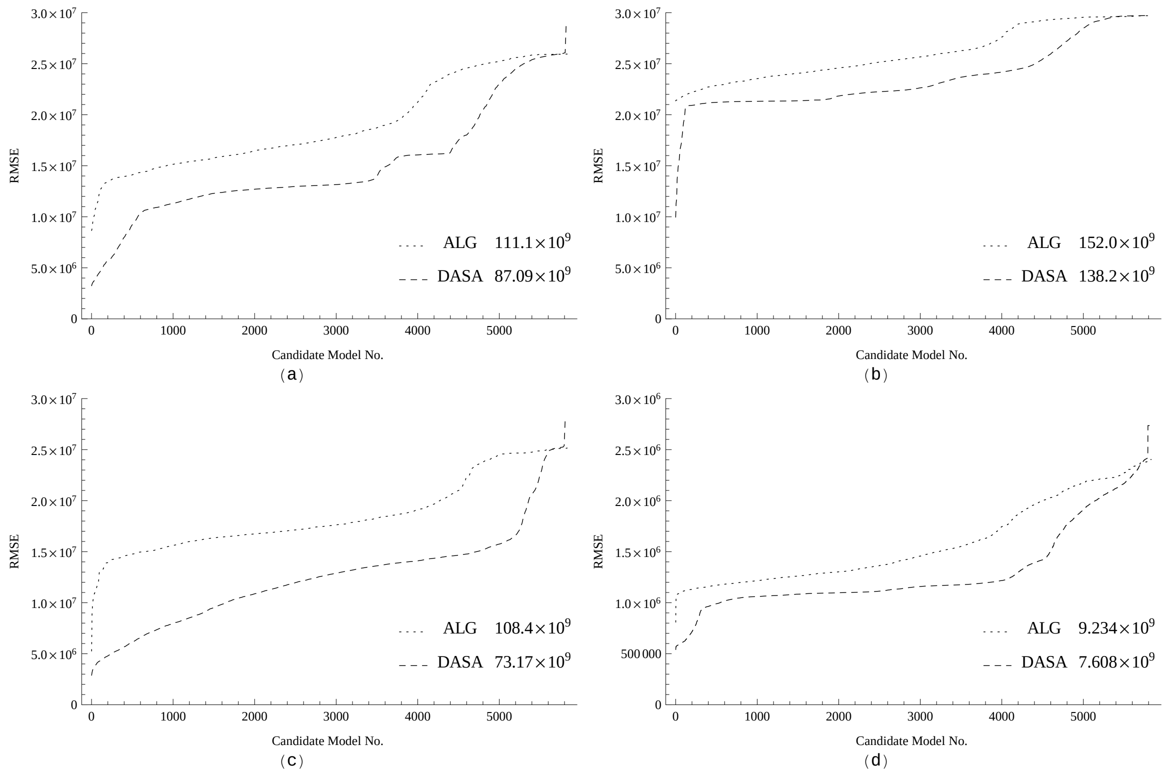


Figure C.6: Error profiles for ProBMoT/ALG and ProBMoT/DASA on all automated modeling tasks for the Venice domain. (a) Venice Loc. 0; (b) Venice Loc. 1; (c) Venice Loc. 2; (d) Venice Loc. 3.

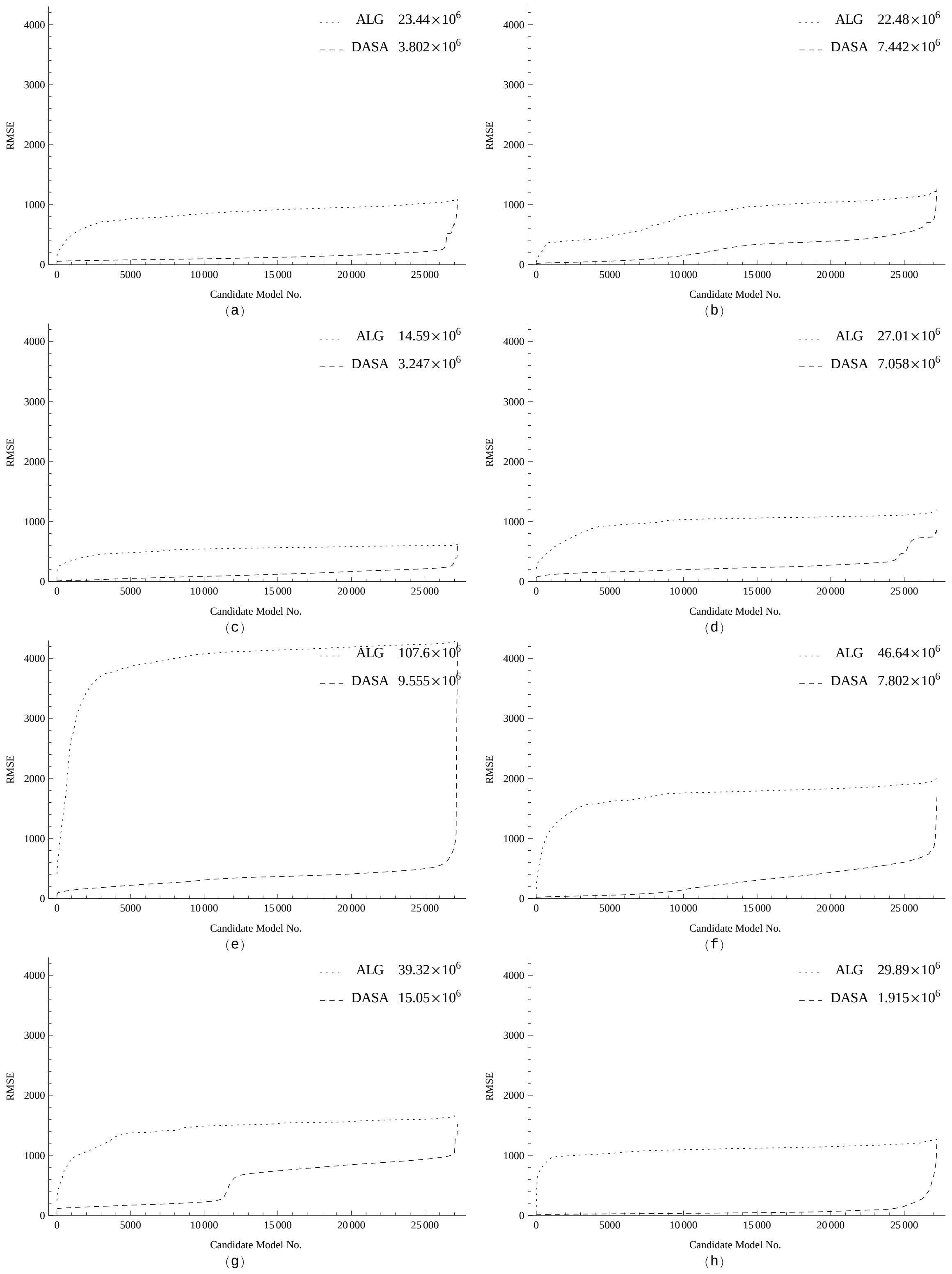


Figure C.7: Error profiles for ProBMoT/ALG and ProBMoT/DASA on all automated modeling tasks for the Bled domain. (a) Bled '95; (b) Bled '96; (c) Bled '97; (d) Bled '98; (e) Bled '99; (f) Bled '00; (g) Bled '01; (h) Bled '02.

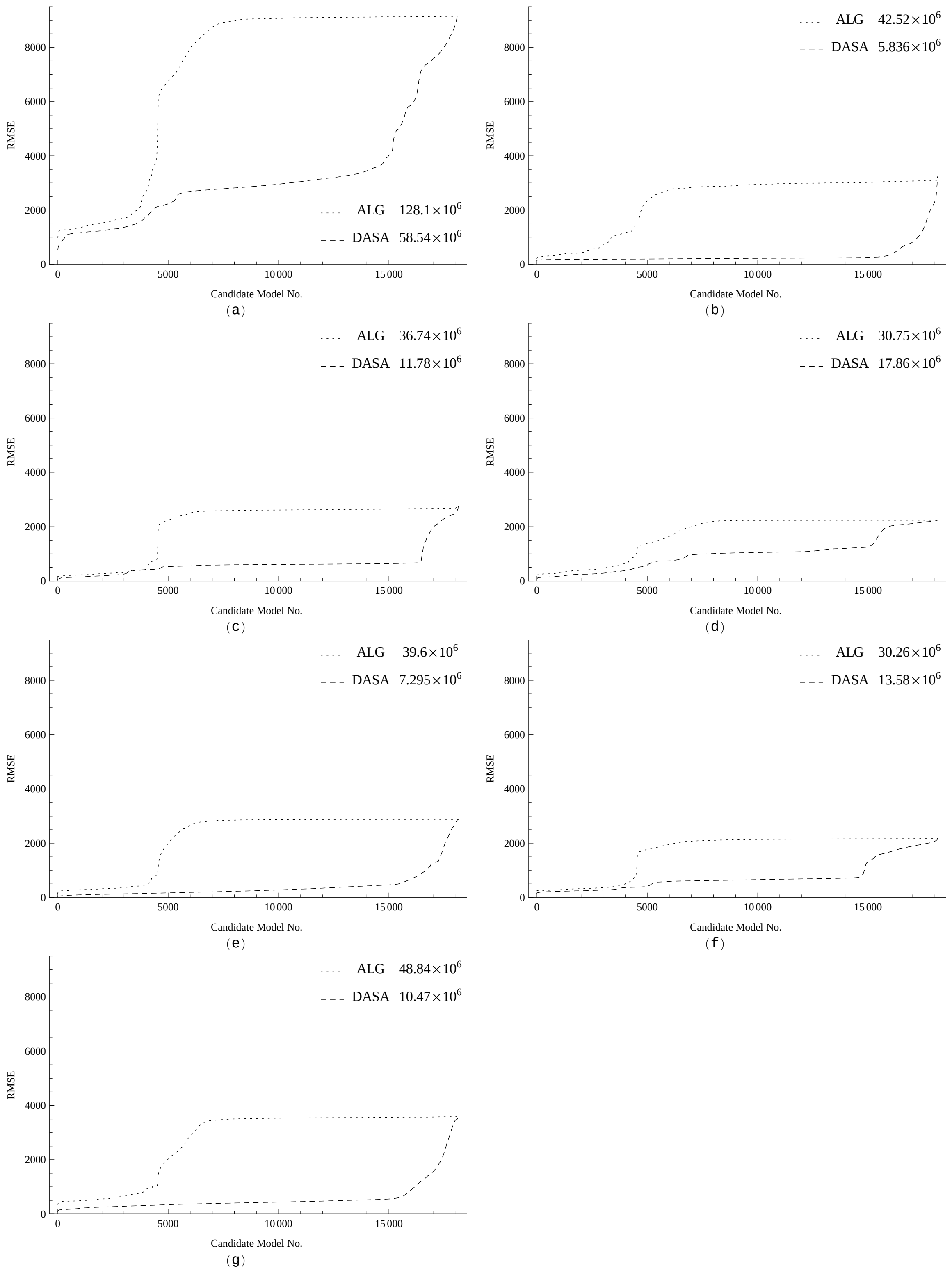


Figure C.8: Error profiles for ProBMoT/ALG and ProBMoT/DASA on all automated modeling tasks for the Kasumigaura domain. (a) Kasumigaura '86; (b) Kasumigaura '87; (c) Kasumigaura '88; (d) Kasumigaura '89; (e) Kasumigaura '90; (f) Kasumigaura '91; (g) Kasumigaura '92.

C.3 Error Difference Distributions

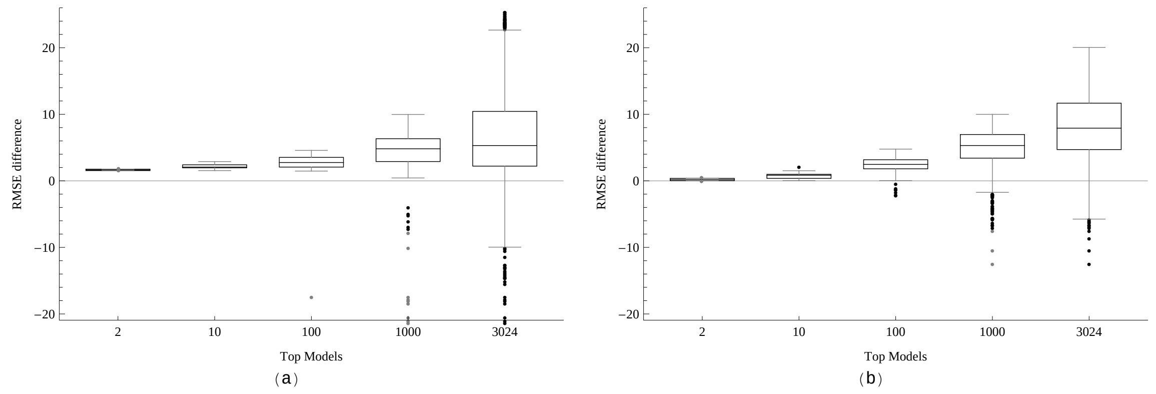


Figure C.9: Distribution of the model-wise differences of RMSE between ProBMoT/ALG and ProBMoT/DASA for all automated modeling tasks for the Glumsø domain. (a) Glumsø ’73; (b) Glumsø ’74.

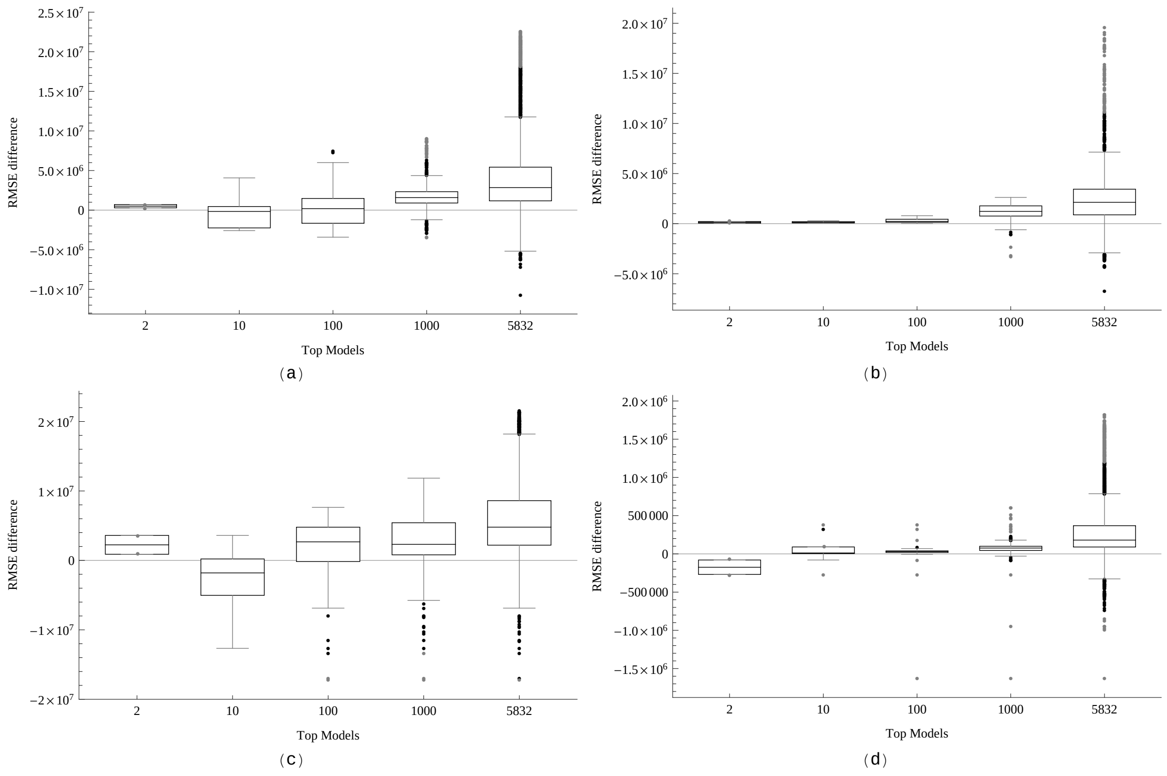
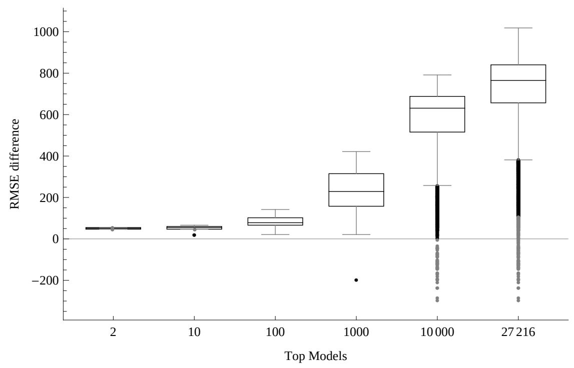
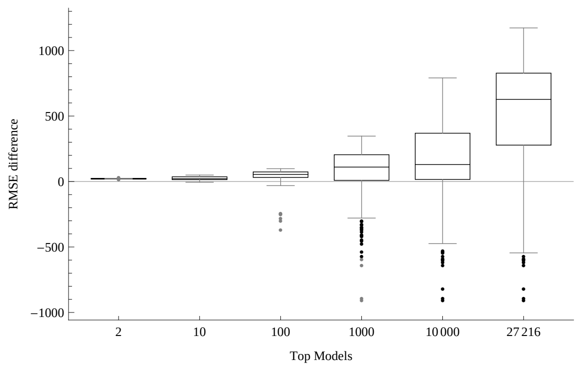


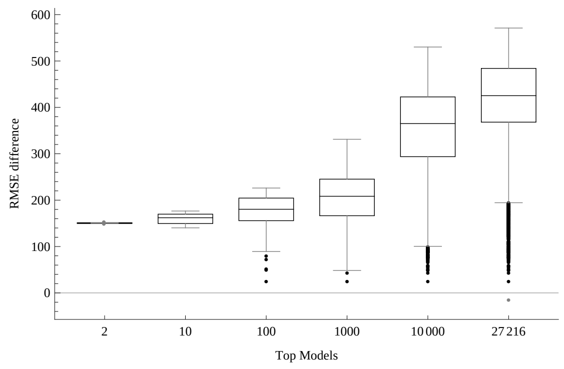
Figure C.10: Distribution of the model-wise differences of RMSE between ProBMoT/ALG and ProBMoT/DASA for all automated modeling tasks for the Venice domain. (a) Venice Loc. 0; (b) Venice Loc. 1; (c) Venice Loc. 2; (d) Venice Loc. 3.



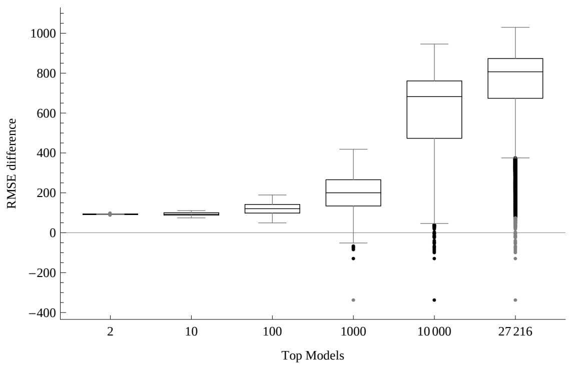
(a)



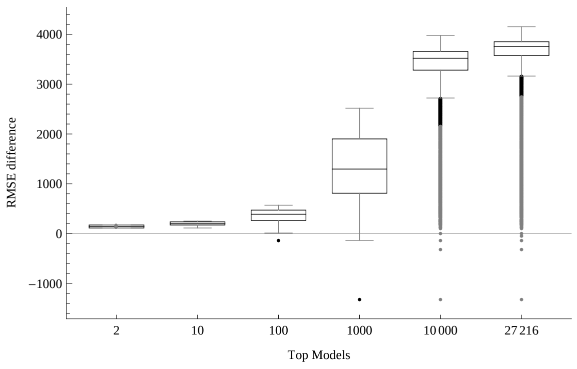
(b)



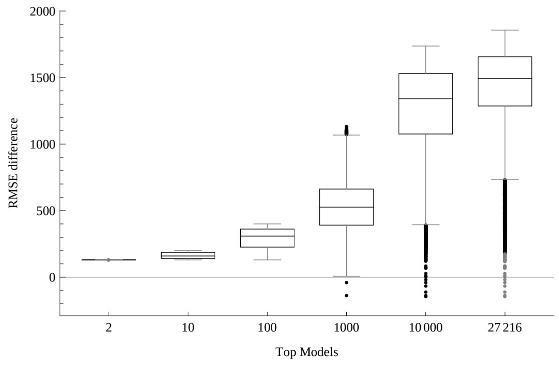
(c)



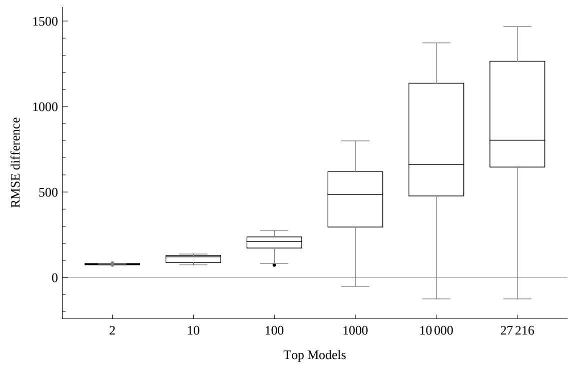
(d)



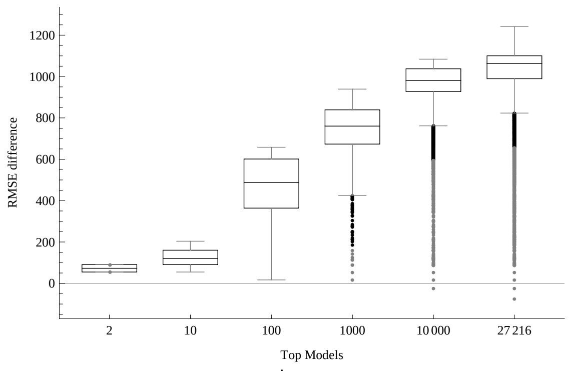
(e)



(f)

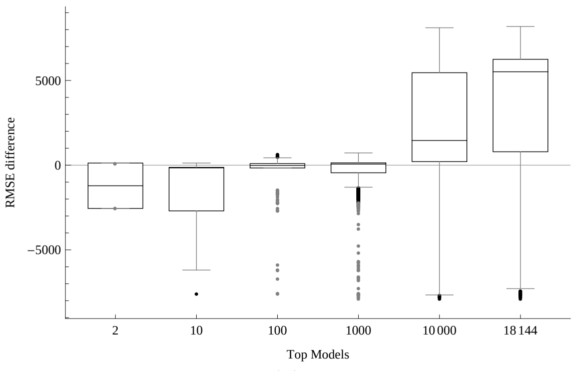


(g)

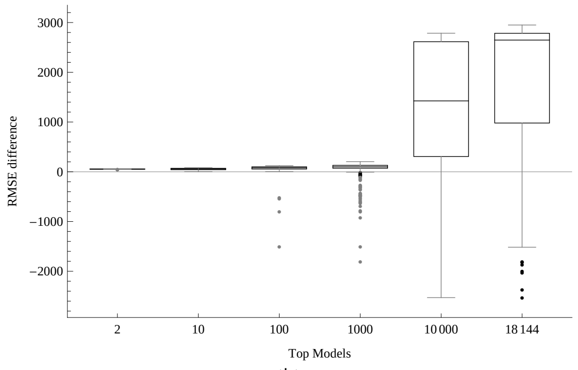


(h)

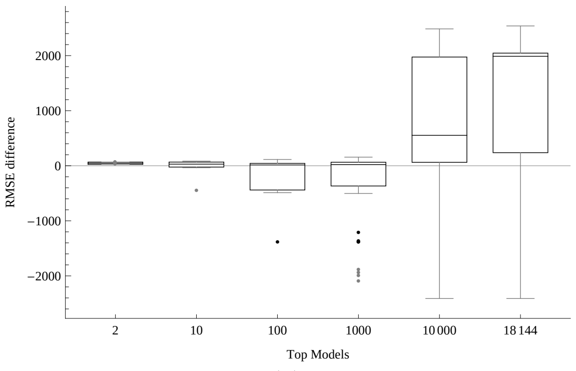
Figure C.11: Distribution of the model-wise differences of RMSE between ProBMoT/ALG and ProBMoT/DASA for all automated modeling tasks for the Bled domain. (a) Bled '95; (b) Bled '96; (c) Bled '97; (d) Bled '98; (e) Bled '99; (f) Bled '00; (g) Bled '01; (h) Bled '02.



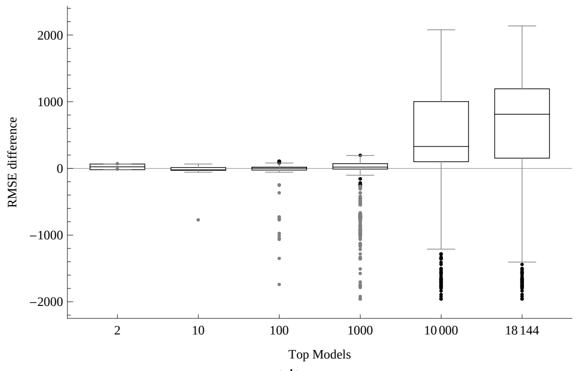
(a)



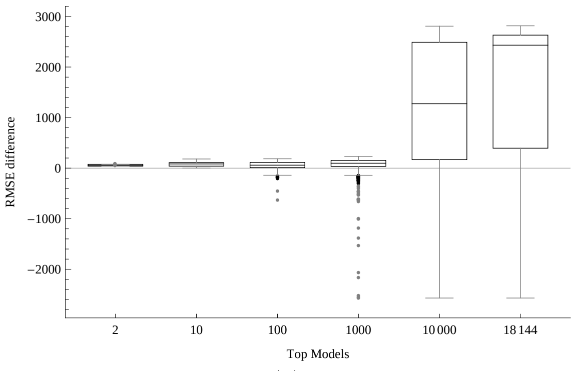
(b)



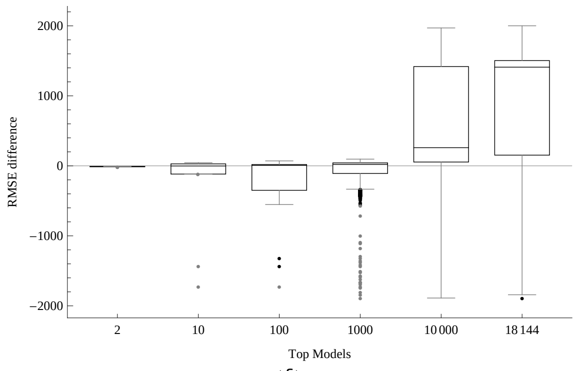
(c)



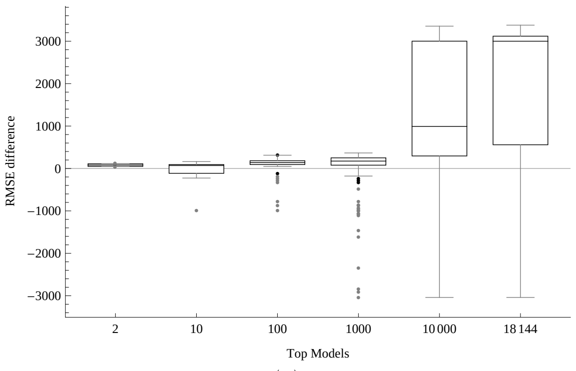
(d)



(e)



(f)



(g)

Figure C.12: Distribution of the model-wise differences of RMSE between ProBMoT/ALG and ProBMoT/DASA for all automated modeling tasks for the Kasumigaura domain. (a) Kasumigaura '86; (b) Kasumigaura '87; (c) Kasumigaura '88; (d) Kasumigaura '89; (e) Kasumigaura '90; (f) Kasumigaura '91; (g) Kasumigaura '92.

D Biography

Darko Čerepnalkoski was born on February 1, 1984 in Skopje, Macedonia. He attended primary and secondary school in Skopje, where he completed gymnasium focusing on natural sciences and mathematics. In 2002, he started his studies at the Institute of Informatics, Faculty of Natural Sciences and Mathematics, Ss. Cyril and Methodius University in Skopje. He was enrolled in a BSc program in the area of Computer Science and Software Engineering. He finished his studies and defended his BSc thesis in 2007 under the supervision of Professor Margita Kon-Popovska. During his undergraduate studies he held a state scholarship for talented students, awarded by the Ministry of Education and Science of Republic of Macedonia.

In the Fall of 2007, he started his graduate studies at the Jožef Stefan International Postgraduate School, Ljubljana, Slovenija. He enrolled in the PhD program entitled *New Media and e-Science* under the supervision of Professor Sašo Džeroski and co-supervision of Professor Ljupčo Todorovski. During his PhD studies, he was a research assistant at the Department of Knowledge Technologies at the Jožef Stefan Institute, Ljubljana, Slovenia.

His research is in the fields of machine learning and data mining and includes the study, development, and application of machine learning and data mining algorithms. His current research—presented in this thesis—aims at developing methodology for inductive process modeling, including formalism for representing process-based models and a platform for learning such models from data and domain knowledge.