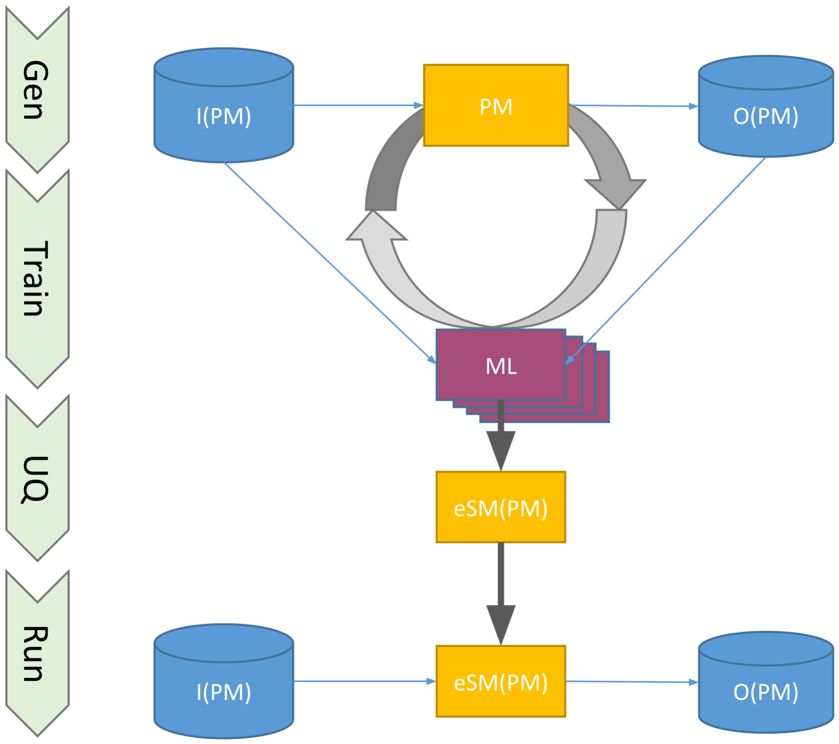
UNIVERSITY OF TRENTO

Doctoral School in Civil, Environmental and Mechanical Engineering

Curriculum 1. Civil and Environmental Engineering

Francesco Serafin

Enabling modeling framework with surrogate modeling capabilities and complex networks



2019 - Doctoral thesis

University of Trento
Doctoral School in Civil, Environmental and Mechanical Engineering
http://web.unitn.it/en/dricam
Via Mesiano 77, I-38123 Trento
Tel. +39 0461 282670 / 2611 - *dicamphd@unitn.it*

UNIVERSITY OF TRENTO
Department of Civil, Environmental and Mechanical Engineering

Enabling modeling framework with surrogate modeling capabilities and complex networks

Dissertation

submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in Civil and Environmental Engineering

by

Francesco Serafin

Supervisor:	PhD Prof Riccardo Rigon
Co-Advisor:	PhD Olaf David

2019

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LISTINGS

ACRONYMS

AgES	AgroEcoSystem
AgES-W	AgroEcoSystem-Watershed
ANN	Artificial Neural Network
API	Application Programming Interface
BFS	breadth-first search
BSON	Binary jSON
CCA	Common Component Architecture
CCP	Cloud Computing Platform
CE	Cellular Encoding
CI	continuous integration
CSIP	Cloud Service Integration Platform
CSU	Colorado State University
CPU	Central Processing Unit
CV	Cross Validation
DAG	Directed Acyclic Graph
DEM	Digital Elevation Model
DFS	depth-first search
DoE	design of experiments
DOI	Digital Object Identifier
DSL	Domain Specific Language
DSS	Decision Support System
EMF	Environmental Modeling Framework
eSM	ensemble of surrogate models
ESMF	Earth System Modeling Framework
ESP	ensemble streamflow prediction
FD-NEAT	Feature Deselective NEAT
FeNS	Framework enabled NEAT-based Surrogate modeling
FICUS	Framework for Integrating the Complexity of Uncertain Systems
FIFO	First-In-First-Out
FS-NEAT	Feature Selective NEAT

GA	Genetic Algorithm
GIS	Geofraphic Information System
GoF	goodness of fit
GoG	Graph of Graphs
GMS	Graph Modeling Structure
GPL	General Purpose Programming Language
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HRU	hydrological response unit
HTTP	HyperText Transfer Protocol
JAMI	Just Another Model Interpolator
JDK	Java Development Kit
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LIFO	Last-In-First-Out
LOC	Lines of Code
LOOCV	leave-one-out cross-validation
MA	Microservice Architecture
MaaS	Model as a Service
ML	Machine Learning
MLPs	Multilayer Perceptrons
MPI	message passing interface
MSE	Mean Squared Error
NE	NeuroEvolution
NEAT	NeuroEvolution of Augmenting Topology
NS	Nash-Sutcliffe
OMS	Object Modeling System
PDGP	Parallel Distributed Genetic Programming
POJO	Plain Old Java Object
PRMS	Precipitation-Runoff Modeling System
PSO	Particle Swarm Optimization
RAM	random-access memory
REST	REpresentational State Transfer

RMSE	Root Mean Squared Error
ROA	Resource-Oriented Architecture
ROTO	Routing Outputs to Outlet
RRS	Reproducible-Research System
RTE	Run Time Environment
RUG	Regional Urban Growth
RUSLE2	Revised Universal Soil Loss Equation, Version 2
SaaS	Software as a Service
SE	Software Engineering
sGA	Structured Genetic Algorithm
SFIR	South Fork Iowa River
SM	Surrogate Model
SOA	Service Oriented Architecture
SoC	separation of concerns
SSoM	System of Systems of Models
SVD	Singular Value Decomposition
SWA	Southfork Watershed Alliance
SWAT	Soil & Water Assessment Tool
SWE	Snow Water Equation
SWMM	Storm Water Management Model
SWRBB	Simulator of Water Resources in Rural Basin
TDD	Test-Driven Development
TIN	triangulated irregular network
TRANSIMS	TRansportation ANalysis SIMulation System
TV	training+validation
TWEANN	Topology and Weight Evolving Artificial Neural Networks
WPS	Web Processing Services
UML	Unified Modeling Language
URI	Uniform Resource Identifier
USDA	United Stated Department of Agriculture
USDA-ARS	United States Department of Agriculture - Agricultural Research Service
USDA-NRCS	United States Department of Agriculture - Natural Resources Conservation Service
VCS	Version Control System

ABSTRACT

Conceptual and physically based environmental simulation models as products of research environments efforts became complex software over time in order to allow describing the behaviour of natural phenomena more accurately. Results from these models are considered accurate but often require to operate an entire system of modeling resources with dedicated knowledge, an extensive set up, and sometimes significant computational time. Model complexity limits wide model adaptation among consultants because of lower available technical resources and capabilities. However, models should be ubiquitous to use in both research and consulting environments.

This dissertation aims to address and alleviate two aspects of research model complexity: 1) for researchers, the model design complexity with respect to its internal software structure and 2) for consultants, the model application complexity with respect to data and parameter setup, runtime requirements, and proper model infrastructure setup. The first contribution provides modeling design and implementation support by managing interacting modeling solutions as "Directed Acyclic Graph", while the second one helps to create surrogate models of complex physical models as a streamlined process.

Both contributions are implemented within the Object Modeling System (OMS)/Cloud Service Integration Platform (CSIP) modeling framework and infrastructure and were applied in various studies.

First, a Machine Learning (ML)-based surrogate model approach is presented to respond to field application requirementes to get quick but "accurate enough" model results with limited input and limited a-priori knowledge of the internal physical processes involved. The surrogate model aims to capture the behaviour of a physical model as an ensemble system of Artificial Neural Network (ANN). Here, the NeuroEvolution of Augmenting Topology (NEAT) technique has been leveraged because of its integration of a genetic approach to build and evolve its ANNs during supervised training. Throughout this phase, the thorough design of the services facilitate seamless monitoring of structural mutations of the artificial neural network and its performances with respect to behavioural emulation of the original model response. This results in a streamlined surrogate model generation. Furthermore, the stochasticity inherent to the evolutionary genetic algorithm combined with a specially designed cross-validation approach allows for straightforward use of the ensemble application. Several, slightly different artificial neural networks are concurrently trained. The ensemble system is built upon the selection of the utmost performant surrogate models and is used collectively to provide uncertainty quantified results when applied against new data.

Secondly, a Directed Acyclic Graph (DAG) modeling structure NET3 was developed. NET3 provides appropriate data structures to represent modeling states interactions as relationships based on network topologies. The inherent structure of the DAG commands the execution of modeling tasks. NET3 implicitly manages the parallel computation depending on the network topology. A node of a NET3 modeling structure encapsulates any sort of modeling solution such as a system of ordinary differential equations, a set of statistical rules, or a system of partial differential equations. Each link connects these modeling solutions by handling their

data flow. As a result, NET3 simplifies 1) the translation of physical mathematical concepts into model components, and 2) the management of complex interactions of modeling solutions. NET3 also pushes forward the idea of separating concerns between software architecture and scientific model codebase. It manages aspects that relate to the architectural design of the graph modeling structure and lets research scientist focus on their model's domain. NET3 improves encapsulation and reusability of scientific/mathematical concepts. It avoids code duplication by allowing the same modeling solution to be adopted in different nodes and finely adapted to specific requirements. In summary, NET3 enables a new level of modeling flexibility by allowing to quickly change model representations to explore new modeling solutions.

The two presented contributions were integrated into the OMS/CSIP Environmental Modeling Framework (EMF)/Cloud Computing Platform (CCP). EMFs are standard practice in environmental modeling because they represent a software solution of separating the burden of software architectural design management from scientific research.

Here, OMS/CSIP has been identified *"advanced"* in terms of EMFs design. It offers high flexibility, low language invasiveness, fine and thorough architectural design, and innovative cloud computing deployment infrastructure. These aspects make OMS/CSIP infrastructure the suitable platform to host NEAT based surrogate modeling and NET3 extensions. Framework enabled NEAT-based Surrogate modeling (FeNS) results from the full integration of NEAT based surrogate modeling approach with OMS/CSIP platform. Here, the surrogate model approach was developed as CSIP services to help transitioning from research models to *"field models"* by enabling the modeling framework to interact with CSIP services, ML libraries, and a NoSQL database to emerge model surrogates for a(ny) modelling solution. OMSCSIP was extended to harvest data from each model run and automatically derive the surrogate model at the modeling framework level. NET3 extends OMS modeling simulations to run as a graph network of interconnected modeling solutions. Furthermore, it enhances available OMS calibration algorithms to become multi-site calibration procedures. OMS already provided implicit parallel computation of independent components in a modeling solution. NET3 now adds a further layer of implicit parallelism by concurrently running independent modeling solutions.

Two studies were carried out to develop and test FeNS while three applications supported the development and testing of NET3.

Surrogate models of the Revised Universal Soil Loss Equation, Version 2 (RUSLE2) were generated to scale up from simple test cases with a constrained input space to more generic applications including a larger variety of input parameters. The main goal of the surrogate model was to streamline and simplify access to the RUSLE2 model behaviour. We performed sensitivity analysis of RUSLE2 to limit the input space to only relevant parameters (e.g. soil properties, climate parameter, field geometries, crop rotation description). The main study area was the State of Iowa starting from a single county (Clay county) ending up to four counties (Buena Vista, Cherokee, Clay, and Wright). Clustering methodologies were applied to improve surrogate model accuracy and to accelerate the training process by reducing the dataset size. The overall "goodness-of-fit" against the testing dataset estimated on the median of the uncertainty quantified result of the surrogate models ensemble was always above 0.95 Nash-Sutcliffe (NS), Root Mean Squared Error (RMSE) between 0.13 and 0.36, and bias between -0.07 and 0.02. In many cases, accuracy of the surrogate model with respect to testing dataset was above 0.98 NS.

Surrogate models of the AgroEcoSystem (AgES) were generated to apply and test FeNS methodology to a semi-distributed hydrologic model. The main goal of the surrogate model was to streamline and simplify access to the AgES model behaviour. Only relevant lumped parameters on watershed centroid were used to train the surrogate models and limit the input space to only relevant parameters (e.g. precipitation, groundwater level, LAI, and potential evapotranspiration). The main study area was the South Fork Iowa River (SFIR) watershed in the State of Iowa across Wright, Franklin, Hamilton, and Hardin counties. The overall *"goodness-of-fit"* against the testing dataset estimated on the median of the uncertainty quantified result of the surrogate models ensemble was above 0.97 NS, RMSE of 2.24, and bias of -0.0794.

With respect to NET3, the first application is the real-time modeling of flood forecasting through GEOframe system for the Civil Protection of Regione Basilicata implemented by PhD Bancheri. To scale the computation and finely tune calibration parameters, the Basilicata river basins were split into subcatchments where each was represented by a different NET3 node.

The second application was part of Mr. Dalla Torre's master's thesis where the computational core of the rainfall-runoff model of Storm Water Management Model (SWMM by EPA) was componentized. NET3 now allows for reimplementing a concise and lightweight SWMM modeling core and highly parallel model runs. Software architectural design of rainfall-runoff, routing and sewer pipe design components targeted separation of concerns, single responsibility, and encapsulation principles. It resulted in clean and minimized code base. NET3 manages component connections and scalable computation by hosting rainfall-runoff modeling solution into separated nodes from routing and sewer pipe design modeling solution. It also enables each node of the modeling structure to 1) access a shared data structure to fetch input data from and push results to (SWMMobject), and 2) internally analyze the upstream subtree in order to adjust sewer pipe design parameters.

The third test case is the application of a System of Systems of Models (SSoM) where each node of the graph modeling structure encapsulates a single responsibility system of urban models. Because of the stochasticity involved in each system of models, the entire graph modeling solution was required to run several times and generate independent realizations. Hence, NET3 was enabled to run a Graph of Graphs (GoG) modeling structure.

1 INTRODUCTION

Contents

1.1 PROBLEM STATEMENT

Conceptual and physically based environmental simulation models as products of research environments efforts became over time complex software that allow for accurately describing the behaviour of natural phenomena. However, on-the-field personnel and consultant agencies struggle to properly exercise these models because of their steep learning curve and excessive runtime requirements. Additionally, scientists themselves strive to maintain and improve such modeling software mostly because the development lacks proper software architecture design and application of good programming principles.

The development history of a model usually starts from a core mathematical concept codified into a piece of software for solving one dedicated problem. Over the time, the need of accounting for more simultaneous physical processes, describing and studying natural phenomena at different scales or introducing innovative engineering design practices drives model development by expanding functionalities and capabilities.

The increased complexity goes usually along with a higher number of input parameters and datasets, and more complicated numerical methods for solving coupled differential equations (Formetta et al. (2014a)). Research advancements in modeling or mathematical fields and technological progress resulting in higher computational power and high resolution data availability fuel the need for models representing environmental reality at different scale more accurately.

This evolutionary process mutates the initial single responsibility code base into multi-responsibilities model: its core gets expanded with additional computational modules, subroutines or even tools for managing and homogenizing a wide variety of input datasets, model parameters and model structures. The mathematical model becomes over time a mix of multidisciplinary tools which have to be maintained and developed. Advancements in each tool have to be coordinated and integrated with advancements in every other related tool. For example, between 1980s and 1990s GIS algorithms and capabilities got to the point of proved stability and robustness and GUIs facilitated user interaction to perform complex geographical analysis (Brovelli (2006)). In the following years, as reported by Westervelt (2001) some famous modeling softwares such as AGNPS (Young et al. (1989)), ANSWERS

(Beasley et al. (1980)), CASC2D (Julien and Saghafian (1991)), GLEAMS (Leonard et al. (1987)), SWAT (Arnold and Allen (1999)), RZWQM (Team et al. (1998)), WEPP (Laflen et al. (1991)), MODFLOW (Harbaugh et al. (2000)), WAMS (DePinto and Rodgers (1994)) integrated GIS (GRASS (Goran et al. (1983); Ehlschlaeger (1989); Westervelt et al. (1991)) in these specific cases) interfaces and algorithms in their modeling core (Cronshey et al. (1993); Rewerts and Engel (1991); Krummel et al. (1996); Hay et al. (1993); Srinivasan (1992); Srinivasan et al. (1998); Arnold et al. (1995)). Since then, mathematical models and GIS capabilities developed along together. Practical examples are the management and integration of satellite imagery and data assimilation techniques in the standard usage of process-based models (Akinmolayan et al. (2018); Anees et al. (2018); Bayramov et al. (2019)).

In the next section, a deeper analysis of the identified problem is performed. Solid bases and motivations are provided to support the relevance of this research.

1.1.1 Motivations

Conceptual/physical models should be ubiquitous to use in both research and consulting environments. Rizzoli et al. (2006) correctly summarizes model user types and roles (Table 1 is a slightly modified version of Rizzoli et al. (2006)). However, no model actually fits requirement from every user and role simultaneously.

Roles / Users	Hard Coders	Soft Coders	Linkers	Run-ners	Player	View-ers	Providers
Prime						✔	✔
Other End Users				✔	✔	✔	✔
Technical		✔	✔	✔			
Researchers	✔	✔	✔	✔	✔		

Table 1: Modified table from Rizzoli et al. (2006) illustrates matches between model user types (rows) and their roles (columns).

The use of numerical models in both scientific and consultant environments is challenging because of a number of issues. The analysis of these issues motivated this research. However, before going into details of each issue, it is important to understand the meaning of *"operational use"* in research and consultant communities.

Service delivery organizations and consultant companies are mainly end-users of mathematical models. Their goal is to leverage software features to provide stakeholders and decision makers with accurate information in topic like conservation practices (e.g. land management and crop operations to avoid excessive soil erosion), prediction of quantity of interest (e.g. water quantity for electric power plan manoeuvres), etc.

They don't develop or maintain software, they are not capable or interested in improving numerical methods, conceptual design or physical process representations due to lack of expertise and resources. Furthermore, from an IT perspective, they may not have in-house computing environments available to run and deploy modeling software. Consequently, they have to rely on third-party environments and personnel.

In research environments, *"operational use"* means both maintenance/development and application of mathematical models.

Model development involves the integration of last enhancements in conceptual design, or numerical/mathematical and physical fields. Additionally, maintenance of integrated tools like GIS capabilities are an important part of model evolution. IT development involves integration and maintenance of modeling frameworks, db connections and design, and proper development to keep up with last innovations, e.g., cloud computing, scalability on computer clusters and super-computing environments in general.

The application side involves model testing and state-of-art consultancy exercises to solve particular problems. In this case, high expertise is available to deal with complex scientific debugging procedures and results interpretation, input data management and preparation, calibration and sensitivity analysis procedures (Green et al. (2015)). Complete model understanding allows for identifying conceptual, mathematical/numerical problems and modeling inaccuracies and finely tuning input parameters and mathematical aspects, thoroughly testing each and every model capability.

Unfortunately, smaller research groups cannot rely on suitable IT expertise in order to properly design, develop, and maintain big complex models. Past experiences show how lacking of proper software architecture design ends up with chaotic, hardly developable and readable/debuggable code bases (Rizzoli et al. (2006); David et al. (2013); Formetta et al. (2014a)). Which in turn slows down research advancements.

Now that the concept of *"operational use"* in both environments has been introduced and described, it's easier to understand that there are issues related to daily use of conceptual and physical models. And these issues are, nevertheless, different.

Consequently, this problem statement deepens issues analysis in two different subsections. The next subsection identifies and describes issues related to the operational use of mathematical models on the field or in consultant agencies. Subsequently, issues related to operational use of mathematical models in research environments are tackled.

1.1.1.1 *Issues related to the use of mathematical models on the field*

Environmental models are regularly applied by consultant agencies and on-the-field personnel. However, daily use is not effortless due to a series of constraints and issues that are following analyzed.

Rainfall-runoff modelling may serve as an example here. Rainfall-runoff is a highly nonlinear, spatially heterogeneous, and very complex process (Srinivasulu and Jain (2009); Beven (2011)) which is comprised of several different, interconnected processes, some of which are not clearly understood yet (Hrachowitz and Clark (2017); Young and Leedal (2013); Zhang and Govindaraju (2000); Porporato and Ridolfi (2001)). The modeling approach evolved over time from a pure empirical form (the Rational Method is the first empirical model ever published in 1851, developed by Thomas Mulvaney), through conceptual models (first models dates back to 1960s, when simplified equations describing hydrological processes were numerically integrated thanks to increased computational power (Wheater et al. (2012); Beven (2011)), to a fully physically based one (1970s computational power was such to solve partial differential equations (Wheater et al. (2012); Beven (2011); Wagener et al. (2004)). Strengths of empirical models are small input parameter sets required and a fast model runtime. However, they lack result accuracy and physical understanding of involved phenomena. The need of comprehensive understanding of physical processes at different scales pushed research efforts toward the development of conceptual models first and fully physically based consequently.

Conceptual models are built upon a conceptual representation of the analyzed process (Wheater et al. (2012); Beven (2011)). Processes are described by simplified ordinary differential equations, and, in the case of rainfall-runoff models, schematized through interconnected reservoirs (Bancheri et al. (2019)). This allows on-the-field personnel to easily understand the model behaviour. However, the implemented equations involve a number of parameters that are not directly or physically measurable. Thus, complex calibration procedures are necessary to estimate those parameters for a specific catchment. Here, the *"equifinality"* problem arises (Beven (1993)): different combinations of parameter values may fit observed data especially when available data have restricted information content and the performance criterion is based off of a single objective function (Wheater et al. (2012)). Then it is impossible to uniquely identify the model structure and apply it to ungauged catchments. When calibration procedures cannot solve the non-identifiability problem (Beven (1993)), a single set of parameters cannot be estimated. In this case, Generalized Sensitivity Analysis (Spear and Hornberger (1980)) allows to select *"behavioural"* set of parameters according to observed data (Wheater et al. (2012)). Afterwords, the model output is uncertainty quantified and contribution of each input parameter uncertainty evaluated (Wikipedia (2018)).

The equifinality problem has a smaller impact on simulation runs when it comes to multi-criteria optimization. Here, additional information are available and integrated into calibration procedures (Yapo et al. (1998); Wagener et al. (2001, 2000)). Nonetheless, modeling tool-kits support sensitivity analysis in order to investigate and identify the most suitable model structure and parameter uncertainty (Wheater et al. (2012)).

In summary, conceptual models are easily understandable without specific expertise because of their abstract representation of the real world. However, they require both calibration and sensitivity analysis procedures which involve several model runs, setup of design of experiments, and final interpretation of parameter estimate and model outputs.

Physically based or mechanistic models are built upon partial differential equations. The latter are the most accurate mathematical description of physical processes (Wheater et al. (2012); Fatichi et al. (2016)). These equations are discretized as finite difference, finite elements or finite volumes over a spatial mesh and solved numerically (Wheater et al. (2012); Pechlivanidis et al. (2011)). And research in this field is highly active (e.g. Casulli (2017), Tubini et al. (2017), and Dumbser et al. (2019)) (Fatichi et al. (2016); Paniconi and Putti (2015)).

Physical models differ from conceptual models because input parameters are actually state variables (Devia et al. (2015); Fatichi et al. (2016)). These have physical meaning and are actually measurable (Wheater et al. (2012)).

Theoretically these models should be used in ungauged catchments, with input parameters estimated a *priori*. However, this practice is not directly achievable for two reasons: (a) small-scale catchments and laboratory experiments are main sources of the physics underneath these modes and make them not straightly applicables to big catchments; (b) some parameters cannot be evaluated on the entire study area, e.g. spatial heterogeneity of soil stratification. Thus, a comprehensive representation of the study area in terms of input dataset is hardly feasible (Wheater et al. (2012); Pechlivanidis et al. (2011)).

Here, calibration procedures can be useful when some input parameters are unknown, even if they are not mandatory in a mechanistic model workflow. Consequently, sensitivity analysis become useful as well in order to estimate model uncertainty.

Nonetheless, these type of models need a big amount of detailed input information to satisfy initial state requirements. They are really complex and high expertise is required to manage them (Devia et al. (2015)). They require long run-time since a model run basically happens in each computational cell of the grid. Additionally, dedicated supercomputing environments can speed up the computation only if model architectural design account for parallel, or scalable algorithms (Devia et al. (2015)).

The analysis of the different available model types helps to summarize issues and limitations on-the-field personnel and consultant agencies have to deal with in order to apply conceptual and mechanistic models:

1. **Thorough understanding of concerned model:** both type of models require in-depth understanding and expertise. Calibration and sensitivity analysis procedures are fundamental parts in the entire setup of conceptual models. Physics of the involved processes has to be perfectly clear when it comes to setup physical model parameters. Otherwise, wrong usage of tools and parameters setup can lead to incorrect and deceptive results. Planning environments cannot always rely on these expertise.

2. **Data collection and preparation:** big dataset feed both type of models. In conceptual models, they are mostly required for calibration procedures. For mechanistic models instead, spatially distributed initial conditions and study area characterizations are mandatory information. Technology evolution allows to meet these needs by providing satellite data, finer grid raster maps, low-error measuring instruments. However, these information are not available everywhere and, if they are, data preparation and assimilation are not trivial tasks and required dedicated proficiencies and GIS capabilities. Additionally, the lack of input standards which especially concerns old models, calls for the design of model-specific applications to convert raw data into model-compliant inputs. As a result, data collection and preparation end up being a long and tedious operation consultant agencies don't always have time and expertise required to deal with.

3. **Run-time:** both type of model simulations are computationally expensive. Calibration and sensitivity analysis procedures require several conceptual model runs to select the set of parameters that better fits observed data or to estimate model uncertainty. Differently, physically based models are computationally demanding because numerical methods return accurate results on finer grids. Service delivery organizations usually need quick results even if not the most accurate.

   Furthermore, even if models are designed to take advantage of multi-processors machines or computer clusters, service delivery organizations don't usually have physical access to supercomputing environments and cannot rely on in-house proficiency to manage them. Yet, if supercomputing environments are not available, relying on third-party machines and personnel with dedicated expertise is expensive.

In summary, the analysis of issues related to daily use of conceptual or physically based models in planning/consultant environments identifies three main topics: (1) lacking of model understanding and expertise in order to fully manage and leverage model capabilities; (2) the absence of required input dataset for feeding calibration and sensitivity analysis procedures or time to handle and properly convert raw data into model-compliant data; (3) unavailability of proper IT infrastructures to reduce computational runtime or lacking of expertise to apply them.

1.1.1.2 *Issues related to the use of mathematical models in research environments*

Mathematical models result from research environment efforts. Here, "operational use" of these models essentially has two meanings:

1. model maintenance and development, which differs from software maintenance and development since mostly target bug fixing of modelled processes and integration of recent numerical and mathematical enhancements, rather than software architecture refactoring;
2. model testing and application to advanced problems that have often never been addressed before.

In both cases, scientists' approach to research models is not straightforward and the reasons are following described. A paragraph with a dedicated analysis for each previously listed issue is provided. Paragraph 1.1.1.2.1 introduces to EMFs methodology as well, which is used throughout paragraph 1.1.1.2.2.

1.1.1.2.1 Issue experienced during model maintenance and development

Research outputs are always up to state-of-art in terms of scientific content. However, mathematical models historically lack of proper software engineering and have been designed as monolithic code base. The latter has been identified as the main cause of current difficulties in model development and maintenance (Formetta et al. (2014a); Rizzoli et al. (2006); David et al. (2013)). The introduction of EMFs in modeling workflow alleviates source code maintenance and development. Yet, EMFs actually limits modeler creativity.

Monolithic applications constraint collective model development, model sharing and reusability because the code structure lacks of separation of concerns (Martin (2009); Newman (2015)), which means that there are no boundaries between different scientific/mathematical concepts (David et al. (2013); Nadareishvili et al. (2016)). Consequently, a scientists needs deep understanding of the entire model to make modifications and speed up the debugging process (Newman (2015); Nadareishvili et al. (2016)). In terms of deployment into production environment, just a single bug fix or modification of a single line of code of a monolithic application requires the deployment of the entire software (Nadareishvili et al. (2016)). Not to mention the scalability issue: enabling a simple multithreading computation in a monolithic software is complicated already, scalability on computer clusters even more (Newman (2015); Nadareishvili et al. (2016)). As a result, leveraging state-of-art computer hardware solutions becomes a cumbersome and most likely unachievable goal.

There are several reasons why software architecture has always had low priority in environmental model research. Historical, cultural, resource and reward constraints have been identified by David et al. (2013) and are following summarized.

From an *historical* point of view, when the era of model development started, C and FORTRAN were the most notable general purpose programming languages to begin with. These languages rely on free and open source compilers and tools as well as active developer communities. They are procedural programming languages though, which already addresses software development towards monolithic architecture.

From a *cultural* standpoint, environmental modelers usually have self-taught programming knowledge and low expertise in software design and architectural patterns (David et al. (2013); Rizzoli et al. (2006)). Their biggest desire is to dive into deeper and more accurate descriptions of environmental processes, model them, and implement them to test the improved modeling solution (David et al. (2013)). It is obviously not possible and fair to ask a natural resource scientist or engineer to take care of both modeling development and software architecture design.

Here the third constraint comes up. A computer scientist, software engineer or hydroinformatic engineer should take care of properly choosing the most suitable software design by anticipating target architectures to support multi language inter-operable systems, high performance computing environments, and most importantly the refactoring of poorly designed modules (David et al. (2013); Rizzoli et al. (2006)). However, *research budgets* are typically limited and expertise of previous listed professional figures are not usually accounted for.

The last constraint, *rewards perspective*, usually hinges to pure achievement of result accuracy. Model reusability and long-term maintainability frequently have low value and priority consequently.

Eventually, the highest quality scientific model codebase ends up being an handcrafted monolith of thousands of lines of code, which is hard to refactor and redesign (David et al. (2013)).

Several EMFs have been developed in the last decade in order to move the burden of software architectural design apart from pure scientific research. Some of the most notable are: OpenMI (Blind and Gregersen (2005); Gregersen et al. (2007)), Common Component Architecture (CCA) (Bernholdt et al. (2003)), Earth System Modeling Framework (ESMF) (Collins et al. (2005)), Common Modeling Protocol (Moore et al. (2007)), and OMS (David et al. (2013)).

EMFs are built upon the notion of component-based software development. They elevate the principle of separation of concerns by splitting responsibilities between framework and components. A component is a typical object-oriented programming concept where a class or module is the core building block of the entire application (Peckham et al. (2013); David et al. (2013)). The framework enables a software "plug-ins" system where a series of precompiled components can be plugged in or unplugged on necessity . It takes care of runtime component connections via dynamic linking and other complicated tasks such as multi language interoperability, multi-threading parallelization of algorithms, temporal-spatial stepping, etc. (David et al. (2013)).

Accordingly, a monolithic codebase can be refactored into a framework-compliant set of components by extracting the scientific knowledge only and splitting it into single responsibility functional units. Good software design practice identifies the optimum level of component granularity into the encapsulation of a single physical process per module, e.g. evapotranspiration, infiltration, runoff, etc. (Peckham et al. (2013)). Working with higher or lower level of granularity is possible though and might be necessary in specific cases (Qu and Duffy (2007)). However, the "one conceptual/physical process per unit" design identify the most flexible granularity level where scientists can easily swap out an old modeling methodology for an innovative or a more appropriate one. Additionally, interfaces between components and related framework connections become *"physical domain boundaries"* and *"physical fluxes exchange"* respectively, further elevating the concept of modeling natural phenomena.

The benefits of employing an EMF in modeling workflow are streamlined model development, improved developer's software quality and reliability, time and cost effectiveness, high level of modeling flexibility.

Several models are framework compliant already, e.g. GEOframe-NewAGE (Bancheri (2017); Formetta et al. (2011); Formetta (2013); Formetta et al. (2013b,a, 2014a,b, 2016a,c); Abera et al. (2017a,b)), Precipitation-Runoff Modeling System (PRMS) (Leavesley et al. (2005)), AgES and it predecessor AgroEcoSystem-Watershed (AgES-W) (Ascough II et al. (2012); Ascough et al. (2014); Green et al. (2014, 2015)), BioMA (Donatelli et al. (2012)), and TopoFlow (Peckham et al. (2017)). A deeper analysis of GEOframe-NewAGE to establish the upsides of modelling with components is proposed in subsection 2.1.2 *Background work to facilitate operational use of environmental models in research environments*.

Nevertheless, EMFs must facilitate the transition of modeler creativity from mathematical equations into component implementations (David et al. (2013)).

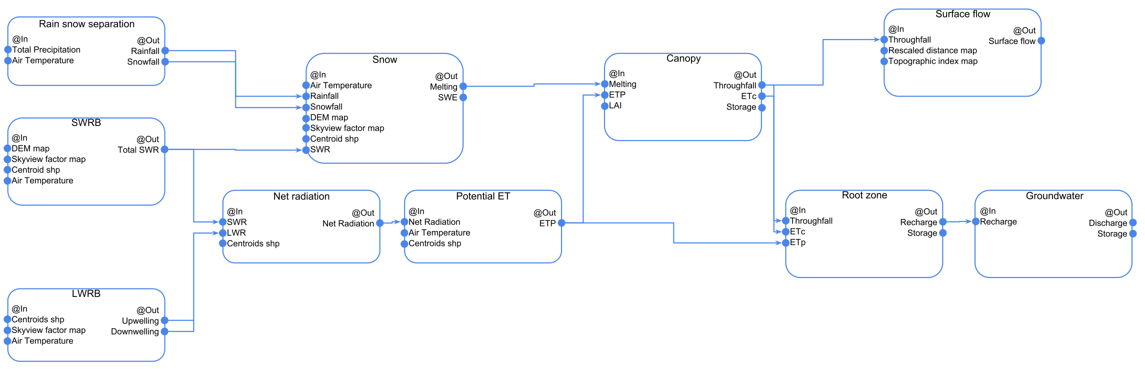


Figure 1: OMS-compliant modeling solution that implements the water budget as theory of embedded reservoir, credit Bancheri (2017).

This lets modeler exploring new problem solving approaches further widening the boundaries of actual modeling techniques.

Figure 1 is used here for the sake of example. It represents the OMS-compliant modeling solution designed by Bancheri (2017) for testing the theory of embedded reservoir model developed during her doctoral research. It estimates the water budget for a generic watershed.

This modeling solution is pretty flexible already: it allows for easily swapping out a single conceptual/physical process, e.g. surface flow, with a different or maybe simpler implementation. However, this modeling approach constraints a scientist to model an entire watershed as an homogeneous entity.

If this is the case of a small watershed, a single modeling solution might match scientist requirements already. It doesn't properly work for a big watershed involving mountain, hill, and plain subcatchments. Here, a modeler might want to finely tune modeling solution parameters for each type of subcatchment. She/he might also want to switch a specific module for a different component per each type of subcatchment, or she/he might want to run completely different modeling solutions in each subcatchment. Additionally, human artifacts such as power plant could potentially become part of a complex modeling solution. But they require dedicated mathematical model and possibly different time loops.

As a result, implementation of complex modeling solutions by leveraging actual frameworks capabilities is not achievable. Actual EMFs functionalities limit modeler creativity and talent. Current EMFs capabilities push back the modeling of complex interconnections and related potential scalable computation to modeler responsibilities.

1.1.1.2.2 Issue experienced during model testing and application to advanced problems

The second issue encountered during regular operational use of mathematical models in research environments relates to actual application to state-of-art consultancy problems.

In this particular case, scientists don't usually have time constraints or lacking of data compared to service delivery organizations. Consequently, application of full mathematical model is achievable and actually fundamental: deep understanding of physical processes allows for studying state-of-art problems and improving modeling techniques eventually.

The constraints are rather on the steep learning curve a scientist has to deal with while approaching a new model. Poor software design and lacking of software coding standards are usually accompanied by deficient source code management.

Research environments are still resistant to adoption of version control systems to track software development, and code peer review is not usually part of standard workflow (David et al. (2013)).

Additionally, a modeler might need to investigate the involved phenomenon from different scales and perspectives, and experiment with modeling solutions consequently. This research methodology requires modeling software to provide high level flexibility to easily accommodate modeler creativity.

1.1.1.2.3 Summary of issues related to the use of mathematical models in research environments

Summarizing, the analysis of issues related to the daily use of conceptual or physically based models in research environments identifies the necessity of adopting EMFs in everyday workflow and the needs of improving their flexibility to:

1. facilitate model maintenance and development;
2. avoid error-prone code duplication;
3. improve modeler productivity and accommodate modeler creativity.

1.2 SUMMARY

The problem statement of this dissertation identifies issues related to the use of mathematical models for (A) service delivery organizations and (B) research environments.

Operational use of conceptual/physical models in service delivery organizations means leveraging research modeling advancements as knowledge encapsulated black-box to support stakeholders and decision makers' questions with accurate predictions and information.

However, consultancy agencies struggle to properly exercise research simulation models since they may lack of (1) expertise to understand conceptual or physical processes requirements to set up calibration or sensitivity analysis procedures; (2) time to collect and prepare datasets to satisfy models high resolution input parameter requirements; (3) in-house availability of computing environments to deploy and exercise modeling simulations.

Operational use of conceptual/physical models in research environments means codebase maintenance, implementation of modeling advancements, and design of modeling simulations for state-of-art consultancy applications. In research environments, a simulation model is a white-box since research scientists have to implement last enhancements in conceptual design, or numerical/mathematical and physical fields. Additionally, scientists need to dig into model implementation when it comes to thoroughly tune calibration procedures and input parameters for advanced applications.

However research scientists struggle to maintain and develop simulation model code base since their are not software engineers, usually have self-taught programming knowledge, and model design commonly lacks of accurate software architecture. Furthermore, current modeling tools constraint modeler creativity and impede the design of innovative modeling solutions.

Issues in both consultant agencies and research environments are getting more and more important since simulation model code base complexity is constantly growing due to integration of research enhancements, engineering practices and additional capabilities such as GIS algorithms. A brief analysis of notable modeling software (Soil & Water Assessment Tool (SWAT) and Storm Water Management Model (SWMM)), which have been developed for more than a decade, is available in Appendix A *NET3: conclusion and future development* to demonstrate this increasing complexity: SWAT overall increment of number of lines of code in about 15 years is more than 100%, while SWMM source grew up of 40% in about 14 years.

The next chapter presents background work and material that attempted to overcome identified constraints. The detailed analysis of current applications of EMFs/CCPs in modeling workflows allows for identifying context and scope of this dissertation, introducing to research contributions proposed to solve identified problems.

2 BACKGROUND WORK AND SIGNIFICANCE

Contents

This chapter introduces to important background information for understanding the significance of this dissertation.

Section 2.1 exhaustively describes previous works in attempt to facilitate operational use of environmental models for service delivery organizations and in research environments. Section 2.2 identifies the modeling core to intervene to, while Section 2.3 points to identified methodologies to expand modeling capabilities and flexibility. Section 2.4 introduces to the actual contributions of this dissertation and describes identified strategies to (1) facilitate access to model behaviour for service delivery organizations, and (2) simplify model development and maintenance and elevate modeler creativity for researchers and modelers. Finally, Section 2.5 summarizes who will benefit from this research.

2.1 BACKGROUND WORK

The identified problems in service delivery organizations and research environments are well established and literature reveals that research work has been done already while attempting to overcome limitations and constraints (David et al. (2013); Peckham et al. (2013)).

This section focuses on analyzing background work that has been conducted so far. Following the structure of this dissertation, background work related to operational use of mathematical models in consultancy agencies is separately described from background work related to operational use of mathematical models in research environments.

2.1.1 Background work to facilitate operational use of environmental models for service delivery organizations

Briefly summarizing, problems that concern service delivery organizations with daily use of conceptual/physical models are mostly related to:

1. the need of thorough understanding of the conceptual schema or physical processes implemented;

2. the requirement of big dataset for calibrations and simple runs;
3. long computational run-time.

The design of complex IT hardware and software infrastructures and the use of surrogate (or complexity reduction) models are two actual attempts to overcome these constraints. This section introduces to benefits and limitations of such problem solving methodologies.

IT infrastructures, supercomputing environments and computer clusters are hardware answers to requirement of higher computational power for reducing model run-time.

The exploitation of these powerful hardware resources necessitates of proper software infrastructures such as EMFs CCPs (Rizzoli et al. (2006); David et al. (2013); Peckham et al. (2013)).

The adoption of EMFs and CCPs carries two additional benefits to service delivery organizations workflow: (1) facilitates encapsulation of simulation models, consequently unifying and simplifying user-model interface by leveraging input/output standards and model-model interface by managing models intercommunication; (2) reduces cost of model development and maintenance by decoupling software architectural design aspects from actual scientific code base (see Figure 2).

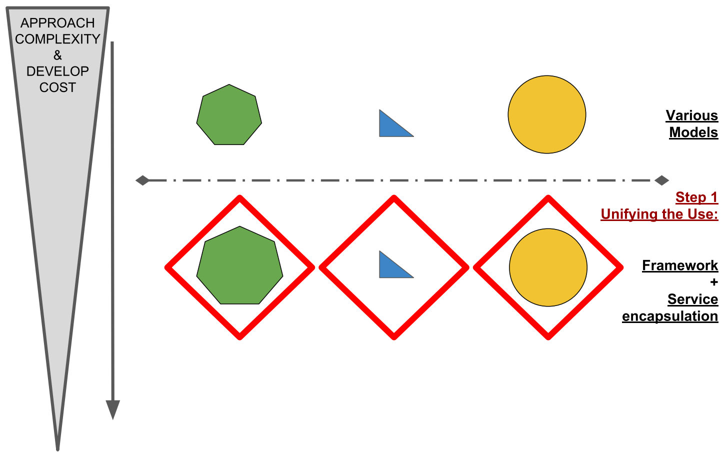


Figure 2: Schematic to represents evolution of research models from stand-alone applications to encapsulated framework-compliant components. This processed unified/simplified user-model interface and model-model intercommunication, and enabled model runs on high performance computing environments. The arrow on the left side illustrates the reduction of model approach complexity by standardizing input/output formats, and the reduction of model maintenance and development cost.

However, if they moderate the run-time problem, facilitate data preparation, and reduce cost expenditure for software development and maintenance, they cannot simplify data-collection operations. The identical input dataset is still required for a framework-compliant or web-service model run.

Additionally, EMFs and CCPs only partially tackle the issue of understanding simulation model internals: they let user take advantage model capabilities as a black box but they don't and cannot simplify calibration and sensitivity analysis procedures.

Databases setup is a further software solution for tackling requirements of big datasets to calibrate and run environmental models. RUSLE2 (Renard et al. (1997); Foster et al. (2000, 2001)) serves here for the sake of example.

RUSLE2 estimates the amount of soil loss along based upon rainfall and related overland flow (Foster (2005)). Several input parameters are required to set up a RUSLE2 model run: from weather conditions, to soil composition and properties, to crop land operations and management impact on soil erosion, etc. To limit the user burden in collecting and setting up input parameters, United States Department of Agriculture - Natural Resources Conservation Service (USDA-NRCS) and United States Department of Agriculture - Agricultural Research Service (USDA-ARS) agencies gathered survey of required input parameters across the entire US into extensive dedicated databases. Consequently, RUSLE2 CSIP-service automatically connects to the proper database at run-time and retrieves user selected parameters.

Figure 3 shows the actual set up of the CSIP-R2 web-service. Four databases are connected directly or through microsubservices to the main R2 CSIP-service. SSURGO contains soil information, LMOD (David et al. (2014b)) contains land management and operations information, while the other two databases provide for remaining parameter requirements.

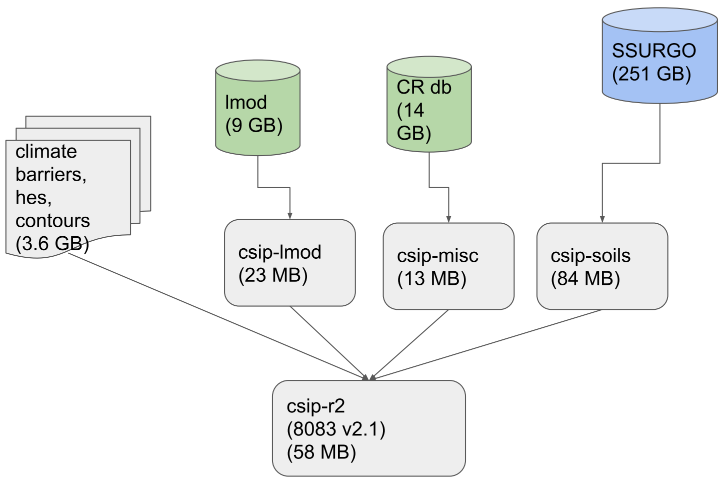


Figure 3: Schematic to represent sub service and database dependencies of CSIP-R2. The overall dependencies are about 290GB in size.

This model setup surely support and ease user approach to RUSLE2. However, it facilitates RUSLE2 application only within US boundaries.

In summary, complex IT hardware and software infrastructures are important background work that served as a first attempt to tackle service delivery organization issues. They simplify some of the inherent complexities of using conceptual/physical models but they:

A. shift the responsibility and workload of model simulation runs to the hosting environment;

B. don't lighten user's burden of data collection, parameter and modeling solution setup;

C. still require model knowledge for handling calibration and sensitivity analysis processes.

Surrogate (or complexity reduction) model is a completely different approach to contemporarily tackle each one of the three problems initially described in this section. These specific types of models are developed to emulate the original simulation model behaviour and speed up the computational time without sacrificing accuracy (Asher et al. (2015)). They were mainly introduced in workflows requiring many simulation runs, such as uncertainty analysis, sensitivity analysis, and optimization frameworks (Beh et al. (2017); Asher et al. (2015); Razavi et al. (2012b)). Here, long runtimes is a big issue and inhibits real time application of complex models, as well.

Three classes of surrogate models exist. Razavi et al. (2012b) proposes a thorough review of surrogate modeling applied to water resources. In a subsequent paper, Asher et al. (2015) identifies, classifies and thoroughly describes the main categories of surrogate models applied to groundwater modeling starting from Razavi et al. (2012b). The resulting taxonomy is following briefly analyzed:

A. **Projection-based models** everage the projection of actual model equations into a basis of orthogonal vectors to reduce the size of the initial vector space. This methodology requires ad-hoc mathematical analysis of each equation implemented in the original model.

B. **Hierarchical or multi-fidelity models** generate from the original models by reducing numerical accuracy, or by ignoring or approximating some physical processes. In the first case, reducing numerical accuracy slightly tackle only the run-time issue. This type of complexity reduction allows for getting pretty accurate results because physical components are identical to the original model. However, the entire input dataset and in-depth model understanding are still required for a simulation set up. The second option slightly tackles the three problems altogether: it speeds up simulation run-time, requires a smaller input dataset, and simplifies the inner modeling complexity. However, these kind of models might still have long computational time, require calibration procedures and consequently model understanding and large input data set. Furthermore, they are not able to return accurate results because it is not possible to compensate simplified physical processes of a system with corrective parameters (Razavi et al. (2012b)).

C. **Data-driven models** (or response surface models) are statistical or empirical models, which are able to capture and approximate the original model behaviour (the response surface, typically a nonlinear hyperplane) by learning the existing nonlinear relationship between a set of original input/output snapshots. There is no direct emulation of any inner conceptual/physical process described in the original simulation model. Literature reports about a large variety of approximation methodologies used for surrogate modeling. In the most recent literature, ANNs are among the utmost commonly used techniques. ANNs are notably flexible function approximators (Razavi et al. (2012b)). However, standard methodologies like Multilayer Perceptrons (MLPs) require several subjective decisions to develop and apply a proper ANN. These decisions involve selection of (1) the optimal structure, (2) the number of hidden layers, (3) the number of neurons in each hidden layer, and (4) the type of transfer function. As a result, surrogate model setup entails an iterative trial-and-error approach (Razavi et al. (2012b)).

In summary, three types of surrogate models have been largely studied and applied in water resources topics. Projection-based and multi-fidelity models originate from actual simplifications of internal conceptual/physical processes of original simulation models, and are tightly coupled with model internals. Thus, these surrogate models require mathematical analysis or ad-hoc configurations for properly emulating each and every simulation model. Oppositely, structures of data-driven models (ANNs in this specific case) are completely decoupled from original simulation models.

This section analyzed background work research community carried out to overcome problems service delivery organizations face in their daily use of conceptual/physical models.

EMFs and CCPs methodologies have been investigated. They facilitate and smooth the approach to mathematical simulation models by (i) elevating capabilities of exploiting powerful state-of-art hardware systems, (ii) unifying user-model and model-model interfaces, and (iii) seamlessly accessing available databases resources. However, they don't solve (A) long computation runs (which management is simply shift to hosting environments); (B) user's burden of data collection, parameter and modeling solution setup; (C) difficulty in handling calibration and sensitivity analysis procedures.

Additionally, surrogate modeling methodology has been investigated. Three types of complexity reduction models have been analyzed and all of them tackle two service delivery organizations problems by reducing computational time and input data requirements. However:

1. projection-based models require ad-hoc mathematical analysis, which are not suited for service delivery organizations requirements;

2. hierarchical or multi-fidelity models might still have long computational time, require calibration procedures and model understanding consequently, and large input datasets;

3. data-driven methodology seems the most appealing approach since its structure is totally decoupled from the original simulation model but standard methodologies still entail an iterative trial-and-error approach.

2.1.2 Background work to facilitate operational use of environmental models in research environments

Briefly summarizing, problems that concern research environments with daily operational use of conceptual/physical models are mostly related to:

1. difficulties in model maintenance and development;

2. error-prone code duplication;

3. lack of model flexibility which limits modeler productivity and creativity.

Research community is aware of challenges, constraints, and frustrations related to monolithic code application maintenance, development, and evolution (Rizzoli et al. (2006); Quesnel et al. (2009); Formetta et al. (2014a)).

The adoption of EMFs in modeling workflows alleviates these problems but it is not widespread among research environments still and sometimes constraints modeler creativity and productivity.

A notable, highly published, state-of-art, framework-compliant hydrological system is GEOframe (Formetta et al. (2014a); Bancheri (2017)). GEOframe serves here for the sake of demonstrating benefits and limitations of integrating EMFs in modeling workflow.

GEOframe system (Bancheri (2017); Rigon et al. (2018)) originates from JGrass-NewAGE project (Formetta et al. (2014a)).

JGrass-NewAGE is a semi-distributed, physically based, OMS compliant hydrological system (Formetta et al. (2014a)). Its development started as a result of Adige River Authority's request for a modeling tool capable of studying drought periods of the river Adige (Rigon (2014)). The entire system results from the interconnection of three groups of software components (Formetta et al. (2014a); Bancheri (2017)):

1. uDig Geofraphic Information System (GIS) is a GIS interface, which allows for geographic data visualization and manipulation;
2. JGrasstools (uDig spatial toolbox) (Abera et al. (2014)), which allows for raster maps management and geomorphological analysis, and includes Horton Machine computing tools (Rigon et al. (2006a,b));
3. NewAGE is a collection of OMS-compliant hydrological modeling components which are following briefly analyzed.

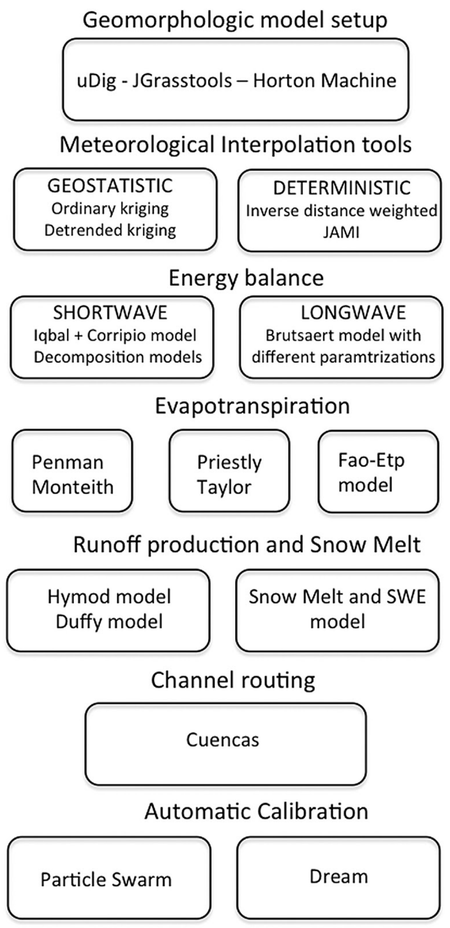


Figure 4: OMS-compliant components designed and developed by Dr. Giuseppe Formetta and released in the initial version of JGrass-NewAGE, credit Formetta et al. (2014a).

Figure 4 from Formetta et al. (2014a) illustrates model components implemented in JGrass-NewAGE. They are grouped into seven categories thoroughly designed by Dr. Giuseppe Formetta. Going through Figure 4 from top to bottom, the first group of software component is the geomorphic and Digital Elevation Model (DEM) analyses, which allow for analyzing raw topographic data and convert it into JGrass-NewAGE input entries. This group of components comprises of raster and vector readers/writers, *Pitfiller* component for filling DEM artificial depressions, *FlowDir* component for estimating flow direction map based off of D-8 algorithm, *HackLength* component for identifying the main stream reach (Rigon et al. (1998)), *ExtractNetwork* component for estimating the channel network based off of drainage direction map, *HachStream* component (Rigon et al. (2006a)) for channel network ordering, *Netnumbering* component (Rigon et al. (2006a)) to couple the hillslope ID with network link it discharges to, and Pfafstetter component for properly numbering the channel network (Formetta et al. (2014a)).

The second group of components is the meteorological interpolation tool, which provides *ordinary* and *detrended kriging* components in addition to *Just Another Model Interpolator (JAMI)* component (Formetta et al. (2014a); Bancheri (2017)).

Radiation forcing is the third group, which makes available *shortwave* and *longwave computational* components (Formetta et al. (2013b, 2014a, 2016a)).

The fourth group relates to evapotranspiration estimate, which provides for three different approaches: *Fao-Evapotranspiration*, *Penman-Monteith*, and *Priestly-Taylor* (Formetta et al. (2014a)).

Runoff production and Snow Melt are the fifth group. Here, two models for runoff estimation are provided: *Hymod* and *Duffy*. Snow melt Snow Water Equation (SWE) is bundled into a dedicated component application (Formetta (2013); Formetta et al. (2013a, 2014b)).

The sixth section includes the channel routing following the Pfafstetter numbering(Formetta et al. (2011); Formetta (2013); Formetta et al. (2014a)).

Finally, the seventh group provides automatic calibration tools such as *Particle Swarm Optimization (PSO)* and *DREAM* (Formetta et al. (2011); Formetta (2013); Formetta et al. (2014a)).

The first version of JGrass-NewAGE demonstrates unprecedented capabilities already: its main author, Dr. Giuseppe Formetta, designed and developed the entire system by providing alternative computational components per hydrological group. Components can be swap out at runtime by leveraging OMS3 framework (David et al. (2013)). JGrass-NewAGE system is highly tested and published (Formetta et al. (2011); Formetta (2013); Formetta et al. (2013b,a, 2014a,b, 2016a,c,b); Abera et al. (2014, 2017a,b)).

Bancheri (2017) enriches the already large variety of modeling components by adding four computational applications: clearness index computation for estimating the incoming and top atmosphere shortwave radiation ratio; net radiation for computing the incoming/outgoing energy balance; embedded reservoir model for conceptualizing water budget throughout interconnected reservoirs; and travel time analysis (Bancheri et al. (2015); Rigon et al. (2016); Bancheri (2017); Bancheri et al. (2017a, 2018a,c); Rigon et al. (2018)).

Despite JGrass-NewAGE high potential and capabilities, Bancheri (2017) reports that early stage modeling components are still influenced by procedural architectural design and lack of software documentation. As a result, Dr. Marialaura Bancheri refactores most of the available components (Bancheri (2017); Bancheri et al. (2018b)) by leveraging object oriented programming notions and design patterns (Gamma (1995); Freeman et al. (2004)). Figure 1 from Bancheri (2017) illustrates the hydrological budget by leveraging newly implemented and redesigned modeling components (Bancheri (2017); Bancheri et al. (2018b)).

Additionally, Bancheri (2017) concretely introduces Reproducible-Research System (RRS) nto JGrass-NewAGE project to consolidate modeling component applications lifecycle across every stage: development, deployment, and production (Bancheri et al. (2016); Bancheri (2017); Bancheri et al. (2017b, 2018a)). Here, the GEOframe organization is founded to define the set of RRS open standards. Consequently, JGrass-NewAGE project is renamed GEOframe system. A software component is GEOframe-compliant if it adheres to the following lifecycle:

1. history of software development maintained with *git* Version Control System (VCS) and regularly committed to GitHub repository hosted at https://github.com/geoframecomponents;

2. components developed as IDE-independent projects by using Gradle building system (Berglund and McCullough (2011));

3. Unit tests serially run at each new VCS commit by incorporating Travis-CI continuous integration in software deployment practice (Beck (2003); Meyer (2014));

4. OMS3 projects for modeling simulation testing deployed at https://github.com/geoframeOMSprojects;

5. modeling components documented according to GEOframe standard and published to http://geoframe.blogspot.com/;

6. tagged modeling components uniquely identified by a Digital Object Identifier (DOI) number leveraging Zenodo archival system.

GEOframe RRS standards are tangible application of David et al. (2013) statement:

> "Adopting software coding standards, using version control systems to manage source code, and performing code peer reviews can be important steps towards improving model code development that will eventually help expedite adoption of modeling frameworks."

Eventually, GEOframe overcomes previously identified JGrass-NewAGE coding practice constraints (Bancheri et al. (2016); Bancheri (2017); Bancheri et al. (2017b, 2018a)) and innovative modeling components are being developed (Bottazzi and Rigon (2018a,b); Tubini et al. (2017, 2018)). Architectural re-design and refactoring of JGrass-NewAGE in addition to application of RRS open standard was possible thanks to its component based structure. This is a very important aspect when it comes to model development and maintenance.

However, Bancheri (2017) still reports modeling constraints with respect to highly heterogeneous systems. Actual OMS3 capabilities allow for simulating a catchment water budget as a whole modeling solution. Finely tuning of modeling parameters for different behavioural domains such as mountain, hill, and plain subcatchments is currently not possible. Figure 5 from Montgomery (1999) illustrates this concept, which is thoroughly described in Montgomery (1999) and presented in Serafin et al. (2018a).

This section analyzed background work research community carried out to overcome problems research scientists face in their operational use of conceptual/physical models. The case of JGrass-NewAGE/GEOframe (Formetta et al. (2014a); Bancheri (2017)) has been described to demonstrate that framework-compliant hydrological systems:

1. foster modeler creativity (run-time swap out of modeling components);

2. facilitate model development and refactoring by encapsulating conceptual/-physical processes description into single responsibility model components;

3. increase publishing rate (potentially one paper per component developed);

4. easily accommodate RRS open standard.

However, EMFs functionalities need to be expanded to not constraint modeler productivity and creativity.

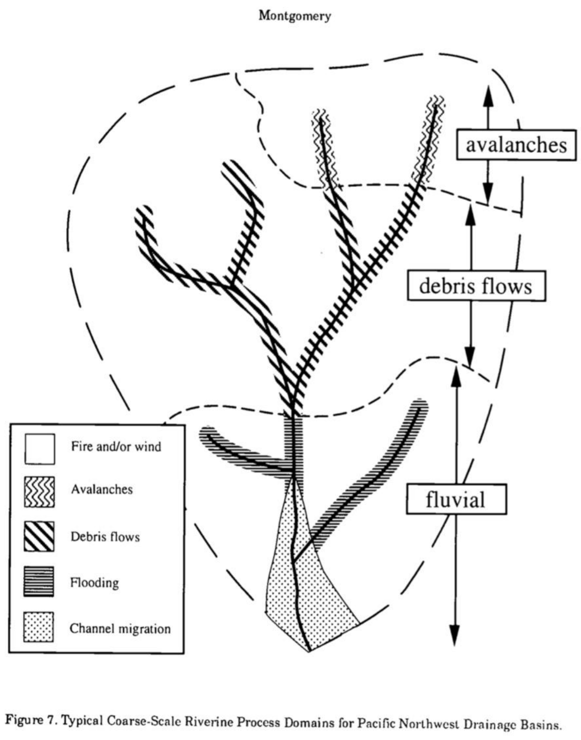


Figure 5: Schematic of watershed scale processes domains in a mountain catchment, credit Montgomery (1999).

2.2 CONTEXT

Although at first glance problems and constraints in both research and consultancy environments seem unrelated, this dissertation identifies a common software core to intervene on and accommodate requirements from both sources.

Strong communication and collaboration between research environments and service delivery organizations historically featured in research advancements and modeling goals achievements. Conceptual/physical models were and still are the intermediaries of this ongoing interaction (Dall'Amico et al. (2018); Bancheri et al. (2018a)).

Problem statement analysis and background work considerations identify EMFs as valuable mainstream tool and proper foster platform where naturally develop innovative modeling practices (Argent (2004); David et al. (2013)).

Figure 6 emphasizes the current modeling architecture. Here, EMFs facilitate and fasten communication between researchers and consultancy environments: researchers utilize EMF platforms as (1) hosting environment for model development and testing and (2) deployment hub of state-of-art modeling applications; service delivery organizations tap into EMF hubs to improve their consultancy capabilities and provide scientifically up-to-date answers to policy, decision makers and stakeholders.

Although problems arising from the two communities are tackled separately, the common solution strategy indicates EMFs as the suitable software architectural layer for hosting modeling methodology development.

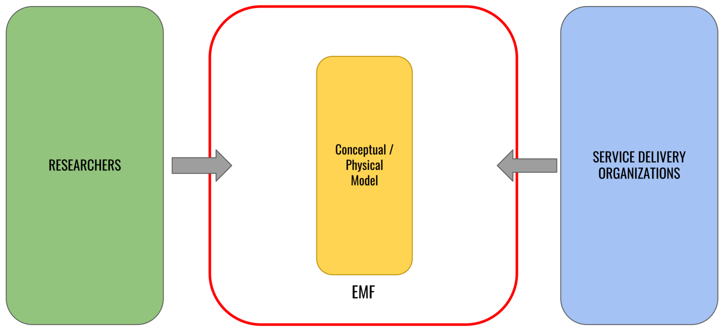


Figure 6: Schematic to represent current modeling practice. Research scientists take advantage of the benefits of EMF architectural design and modeling flexibility to release/deploy as well as access conceptual/physical models; opposingly, service delivery organizations make use of last enhancements in terms of scientific knowledge and modeling practice to provide stakeholders and policy makers with accurate estimate of quantity of interest.

This dissertation locates in EMFs the suitable layer for accommodating research environment and service delivery organization requirements and proposes a set of framework capability extensions to attempt problem solutions in the next section.

2.3 SCOPE

The scope of this dissertation is to expand EMFs and CCPs capabilities to accommodate research environment and service delivery organization requirements. According to the structure of this dissertation, scope of research related to consultancy agencies problems and scope of research related to research environments issues are separately described.

Service delivery organizations need a fast, lightweight, and "accurate enough" tool capable of emulating original conceptual/physical model behaviour with fewer input information. Background work identifies surrogate models (or complexity reduction models) as promising methodology to accomplish this task. Different alternative surrogate models are available in literature, and background work establishes data-driven surrogate model as most appealing approach since its structure is totally decoupled from original simulation model.

This dissertation will research methodological and technical approaches that allows for enabling a modeling framework to interact with ML libraries to emerge data-driven model surrogates a(ny) modeling solution.

The scope of this research topic narrows to the emulation of selected aspects of the original model behaviour. It is surely of high interest to establish a relation between surrogate model and actual/measured data. This is of fundamental importance especially when the estimate of original model is wrong and measured data might compensate model constraints. However, this dissertation won't invistigate this further topic, which is postponed for later research.

Research environments need a proper strategy to overcome actual limitations and constraints research scientists face while developing and maintaining, or simply applying conceptual/physical models. Background work identifies EMFs as state-of-art software environments to foster separation of software architectural aspects from scientific concepts. These software tools already facilitate model development and usage, and elevate modeler creativity, which was previously constrained by monolithic applications. However, problem statement and background work demonstrate how state-of-art modeling solutions start limiting modeler needs.

This dissertation will research methodological and technical approaches that allow for expanding EMFs capabilities in terms of modeling flexibility. This dissertation will investigate the integration of graph theory into EMFs core capabilities to accommodate research scientists requirements of modeling complex network-like interactions by expanding the already flexible EMFs modeling approach.

In conclusion, this dissertation will research methodological and technical approaches for:

1. enabling a modeling framework to interact with ML libraries to emerge data-driven model surrogates a(ny) modeling solution;

2. integrating graph theory into EMFs core capabilities to accommodate research scientists requirements of modeling complex network-like interactions by expanding the already flexible EMFs modeling approach.

2.4 OBJECTIVES STATEMENT

This dissertation contributes to the expansion of actual EMFs and CCPs capabilities with respect to development, maintenance, and access to modeling resources.

To achieve this goal, CSIP/OMS (David et al. (2013); Lloyd et al. (2011, 2012); David et al. (2014a)) has been identified as state-of-art in terms of EMFs to begin with. It has proven to be the perfect fostering environment for further expansions and developments. A higher level of modeling flexibility will be enhanced by elevating its valuable features and already advanced modeling capabilities. Subsubsection 3.3.2.3 *Cloud Service Integration Platform (CSIP)* and 4.4.2.1 *Object Modeling System v3 (OMS3)* investigate and analyze the motivation behind the choice of CSIP/OMS.

This dissertation will develop two framework extensions to accommodate research scientists and service delivery organizations requirements. Figure 7 illustrate both concepts:

1. A modeling layer of surrogate modeling capabilities will be interposed between conceptual/physically based model and modeling users such as consultant agencies, service delivery organizations, and on-the-field personnel to help transitioning from research to field. To emerge data-driven model surrogates a(ny) modeling solution, the modeling framework will be enabled to interact with ML libraries. Here, NEAT (Stanley and Miikkulainen (2002); Whiteson et al. (2005)) will streamline the automated process of model creation. Literature reviews doesn't report any current application of NEAT algorithms in environmental modeling or surrogate modeling methodologies. Finally, this dissertation introduces the concept of FeNS and the protocol that rules framework-ML libraries interaction.

2. The integration of a flexible complex network based graph modeling structure (NET3) into OMS3 software core will extend OMS3 modeling capabilities. This approach will be designed over river network - graph structure analogy. To facilitate model development and elevate modeler creativity, NET3 will connect modeling solutions, provide a further layer of implicit parallelization,and allow for easy implementation of additional features. NET3 will

streamline the transition from forced homogeneous modeling of environmental features to highly heterogeneous and finely tuned environmental modeling.

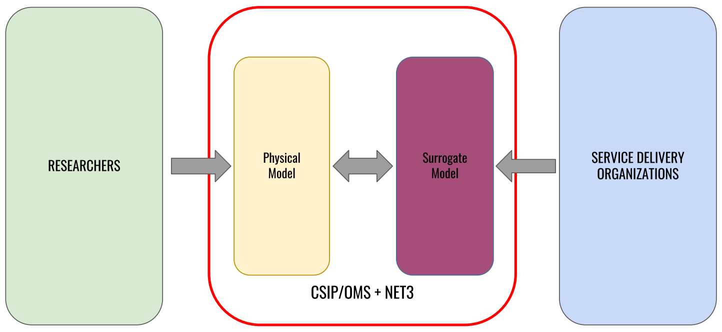


Figure 7: Schematic of actual contributions of this dissertation. A new surrogate modeling layer is interpose to bridge the gap between service delivery organizations and conceptual/physical models. Contemporary, EMF modeling capabilities are extended by implementing a graph modeling structure. This allows for bridging the gap between researcher scientists and modeling platforms by enabling modelers creativity and elevating concept of modeling encapsulation and re-use.

Ensuring that these approaches will completely bridge actual gaps between research environments and conceptual/physical models, and service delivery organizations and conceptual/physical models is out of the scope of this dissertation. However, the enhanced functionalities will facilitate access to mathematical models and create solid foundations for fostering future modeling practice developments.

2.5 RELEVANCE

The findings of this study will redound to the benefit of research scientists and service delivery organizations by facilitating development and access to modeling resources.

Service delivery organizations will be provided with automatically generated surrogate modelling capabilities to facilitate and speed up simulation runs. As a results, running a large variety of modeling scenarios will be faster and less computationally demanding. This potentially fastens support to policy and decision making processes. Additionally, the surrogate model will be a lightweight "detachable" tool that can run from within portable devices on the field with limited information and no internet connection.

Research environments will be provided with innovative EMFs capabilities which will stave off actual modeling simulation constraints. This will result in faster and easier model development and maintenance by allowing for more complex and tunable modeling solutions while promoting code reuse and EMFs-compliant modeling practices. Furthermore, this will facilitate and accommodate creative modeling approaches by allowing for easy implementation of complex network-based modeling solutions.

2.6 SUMMARY

This chapter analyzes background work that drove this dissertation toward identification of context, scope and objectives statement.

This dissertation contextualizes CSIP/OMS EMF/CCP, and defines two research goals to expand its modeling capabilities for accommodating requirements from service delivery organizations and research environments accordingly.

This dissertation will research methodological and technical approaches for enabling a modeling framework to interact with ML libraries to emerge data-driven model surrogates a(ny) modeling solution. This study investigates NEAT algorithms to automatically emerge the model surrogate and defines FeNS protocol to enable the intercommunication between modeling framework and ML libraries.

This dissertation will research methodological and technical approaches for expanding modeling framework capabilities with graph modeling structure. This study investigates graph theory and actual software implementation into modeling platforms that resembles river network – graph structure analogy.

Two distinct chapters describe the two identified approaches by following identical outline: literature review and research questions determine starting point of each study and path that guides through the research respectively, while research methodologies and case studies report methods leveraged to carry out the analysis and their application to actual test cases.

Chapter 3 researches surrogate modeling approaches and their integration into modeling framework, while Chapter 4 discusses the integration of graph modeling structures in the modeling workflow.

3 SURROGATE MODELING

Contents

The problem statement of this dissertation highlights applicability constraints of conceptual and physical models originating from research in consultant and planning environments. Due to their complexity, data resolution requirements, number of parameters, platform affinity, and other criteria mathematical models are rarely suited "out-of the box" or field and consulting applications. A surrogate modeling methodology is proposed to tackle these issues and facilitate the transition from research models to service delivery organizations and consultant agencies requirements. This dissertation proposes a ML-based surrogate model approach aiming to capture the intrinsic knowledge of a mathematical model into an ensemble system of artificial neural networks and make it available for providing simplified answers to on the field problem-specific questions. A surrogate modeling approach was developed to help transitioning from research to field by enabling a modeling framework to interact with ML libraries to emerge model surrogates a(ny) modelling solution. CSIP/OMS was extended and utilized to harvest data and derive the surrogate-model at the modeling framework level. Here, NEAT techniques in an ensemble application, combined with ANN uncertainty quantification are the main methodologies used and following exhaustively described.

3.1 LITERATURE REVIEW

Surrogate modeling is not a new concept in research environments. The first publication dates back to Blanning (1975), where the author conceptualizes the need for surrogate models (named metamodels in that paper) for sensitivity analysis purposes. Kleijnen (1975) rovides statistical tools to make Blanning's theory operational.

Since then, this idea has evolved and largely applied in workflows requiring many simulation runs such as optimization, sensitivity and uncertainty analysis operational management, and prediction (e.g. Viana and Haftka (2008); Razavi et al. (2012b); Asher et al. (2015); Beh et al. (2017)). Here, long runtimes is a big issue and inhibits real time application of complex models as well. Several papers have been published on the topic. Razavi et al. (2012b) and Asher et al. (2015) are the most important review articles on surrogate modeling in water resources.

Asher et al. (2015) reports the usage of different names like *metamodels* (Blanning (1975)), reduced models (Willcox and Peraire (2002)), model emulators (O'Hagan (2006)), proxy models (Bieker et al. (2007)), lower fidelity models (Robinson (2007); Robinson et al. (2008)), and response surfaces (Regis and Shoemaker (2005)). He also provides a detailed taxonomy of surrogate models based off of their mathematical structure:

- **Data-driven methods:** (AKA response surface, statistical and black box method) are empirical approximator surrogates created from a set of model inputs/outputs which emulate high-fidelity model responses;
- **Projection-based methods:** (AKA reduced order, reduced basis and model reduction methods) are generated by creating a basis of orthonormal vectors to reduce dimension subspace where project governing equations (usually Krylov-based and Singular Value Decomposition (SVD) methods);
- **Multifidelity based methods:** (AKA multiscale, hierarchical and physically based methods) result from decreasing numerical resolution or by reducing underlying physics complexity.

The goal of this research is to automatically generate surrogate models at framework level without requiring user extensive intervention. Between the three previously listed categories, the choice of the methodology to use fell into data-driven methods. Here, ANN are the only highly non-linear approximator that can be automatically generated.

Consequently, this dissertation and this literature review focus on response surface surrogates and ANN more specifically.

The following definition allows for understanding the reason why this mathematical tool is named artificial neural network:

> "A neural network is a massively parallel distributed processor that has a natural propensity for storing experimental knowledge and making it available for use.
>
> It resemble the brain in two respects:

1. Knowledge is acquired by the network through a learning process;
2. Interneuron connection strengths known as synaptic weights are used to store the knowledge."

– Haykin and Lippmann (1994) –

A more rigorous mathematical definition describes an ANN as a directed graph (Floreano et al. (2008)). Here each node (or *neuron*), except for the input layer, is the actual processing element: a usually nonlinear static transfer function transforms the weighted sum of input values into a single output value (Govindaraju and Rao (2000b)) (see Figure 8).

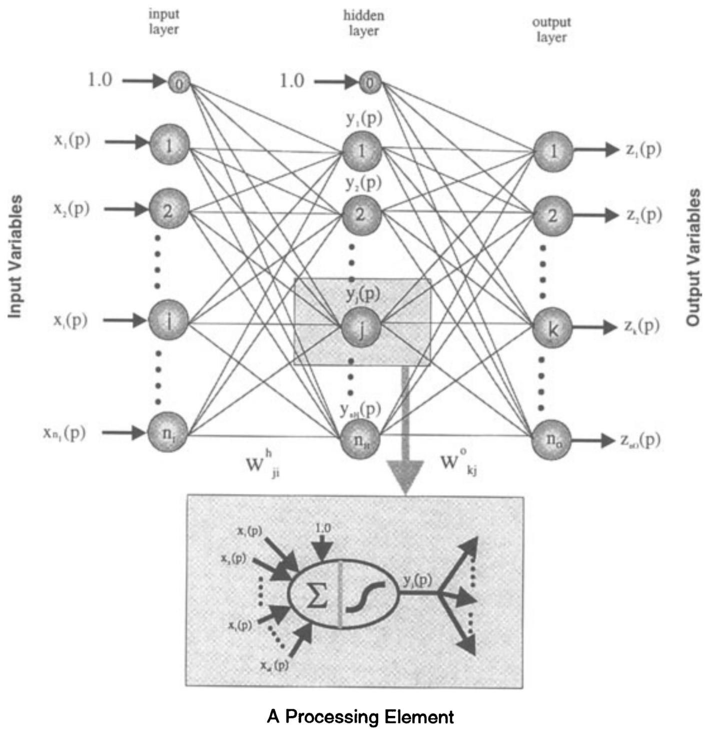


Figure 8: Single hidden layer feedforward neural network, credit Govindaraju and Rao (2000a).

ANN is an empirical data-driven type of models generated by capturing the behaviour of high fidelity model through their input and output datasets.

Over the past 20 years, applications of ANNs (McCulloch and Pitts (1943)) to water related topics for surrogate modeling purposes have taken off thanks to advancements in computational power and parallel distributed environments but more importantly, after the introduction of solid mathematical basis by Hopfield (1982) and Rumelhart et al. (1985).

ANN-based Surrogate Model (SM)s have been used for uncertainty-based automatic calibration studies such as Khu and Werner (2003) (they performed auto-calibration of SWMM model), or Zhang et al. (2009) (they performed auto-calibration of SWAT model), or Zou et al. (2009) (they performed auto-calibration of WASP model).

They have been implemented in multiobjective optimization settings such as Liong et al. (2001) (they performed auto-calibration of HydroWorks model through bi-objective optimization) or Behzadian et al. (2009) (they performed the optimization of water distribution system monitoring locations through bi-objective optimization).

Shrestha et al. (2009) replaces computationally demanding Monte Carlo simulations with ANN-based SMs (they performed prediction of uncertainty estimation of a hydrologic model). Yan and Minsker (2006, 2010) make use of neural networks to solve integer optimization problems (they designed groundwater remediation strategies using MODFLOW and RT3D for flow field and contaminant concentration).

However, Razavi et al. (2012b) emphasize the relevance of identifying the optimal structure of an ANN to properly design ANN-based surrogates, and the fact that this results from researcher subjective decisions and trial-and-error processes.

Additionally, they underline the ANN nature of being inexact or almost exact emulators, which might not fit requirement of approximating deterministic response of a computer model for optimization purposes.

Although ANN-based SM might not perform perfectly away from design sites, Asher et al. (2015) states that

> "Despite their drawbacks, well used data-driven approaches remain a valuable tool in applications such as decision support and integrate modeling, where it may be necessary to limit both the number of parameters and the ranges which they take. Quick runtime once calibrated and their non intrusive nature make data-driven methods particularly useful for these applications."
>
> – Asher et al. (2015) -

This research attempts to overcome current limitations of ANN-based surrogates and targets the automatically emerging of the surrogate model at a framework level in addition to input space dimensionality reduction.

The final goal is to provide service delivery organizations with a lightweight and easy to use tool to facilitate decision support on the field.

3.2 RESEARCH QUESTIONS

This section introduces to research questions this dissertation investigates on. Each research question is briefly analyzed.

RQ1: Can we sufficiently duplicate the behaviour of relevant conceptual or physically based models with abstract generic implementation of surrogate models?

ANN-based SM as methodology itself has been widely applied and proved to return accurate estimates (Kourakos and Mantoglou (2009); Yan and Minsker (2006)). However, the structure of each surrogate results from researcher subjective decisions and trial-and-error processes (multilayer perceptron neural network might have one or more hidden layers and the actual number of hidden nodes is usually consequence of a trial and error procedure).

NeuroEvolution of Augmenting Topology NEAT is an evolutionary algorithm designed to evolve the structure of a neural network starting from the minimal topology and incrementally growing it. This methodology allows for automatically emerging SM with the most appropriate internal structure since topology and weights of the artificial neural network evolve during supervised learning.

RQ2: How can we properly split the input dataset in order to emulate models behaviours more accurately?

Resampling methods are widely applied in modern statistics and are fundamental tools for ANN methodologies as well (James et al. (2013)). Here, cross-validation is mainly applied for:

- **Model assessment:** in order to estimate the performance of a model and avoid overfitting;
- **Model selection:** in case of fixed structure ANN, leave-one-out cross-validation (LOOCV) and k-fold Cross Validation (CV) are mainly used to test ANN with different structures and select the most performant one.

In this research ANN methodologies are utilized as model behaviour approximators. Thus, the relation between model input and output should be described by a hidden but well defined nonlinear function since the dataset is noise-free. As a result, model assessment (overfitting) is not the biggest concern.

Regarding model selection, NEAT takes advantage of the benefits of a genetic algorithm to build the ANN structure during the training phase. Thus, NEAT ends up generating a different ANN every time the training process starts, even if it is fed with the same training dataset. Consequently, standard resampling methods cannot be applied to this research.

However, properly splitting the training dataset to homogeneously cover the domain space and leveraging NEAT aleatoric process to generate several slightly different ANNs might potentially improve SM estimates accuracy.

RQ3: How can we improve surrogate model results accuracy and provide strong support in the decision making process consequently?

One of the main goals of mathematical/engineering models is to facilitate and support decision making processes. This goal is achieved by quantifying the range of possible, which means estimating uncertainty and variability of the model (Swiler and Giunta (2007)). The NEAT methodology involves several aleatoric processes, which might be used to uncertainty quantifying ensemble of ANNs result.

Furthermore, ensemble learning has demonstrated to improve result accuracy with respect to single ANN application (Yu et al. (2008); Guo et al. (2012)).

Combining ensemble learning with uncertainty quantification leads to achieving two goals: improve result accuracy while supporting and ease decision making processes.

3.3 RESEARCH DESIGN AND METHODS

This research aims to support consultant agencies by providing a more lightweight and easier to use surrogate of an actual mathematical model. This surrogate is generated at a framework level by expanding framework capabilities.
Thus, this research has been driven by the need of:

- Automatically generate the surrogate model at framework level;
- Generate a surrogate model able to provide accurate results and support decision makers by uncertainty quantifying surrogate model behaviour.

Subsection 3.3.1 *Methodological approach* introduces to the two main methodologies this research utilizes and the two conceptual contributions:

- **NeuroEvolution of Augmenting Topologies (NEAT)** is the evolutionary algorithm capable of creating the structure of the ANN while adjusting connection weights during the training phase (Stanley and Miikkulainen (2002));
- **Feature Selective NEAT (FS-NEAT)** is a NEAT extension that introduces an implicit dimensionality reduction mechanism to select only input parameters that yield to the best ANN performance during the training phase (Whiteson et al. (2005));
- **Ensemble of SMs coupled to uncertainty quantification of SM results** is the methodology developed to take advantage of FS-NEAT inherent stochasticity and improve result accuracy;
- **Framework-enabled NEAT based Surrogate modeling (FeNS)** is the final concept and contribution of this dissertation. FeNS integrates the three previously introduced methodologies at framework for automatically emerging the SM.

Subsection 3.3.2 *Technical approach and implementation* deeply describes software libraries, Application Programming Interface (API)s, and platforms employed to extend framework functionalities to generate the surrogate model. An introduction to MongoDB database as well as Microservices and RESTful API is proposed. Then, a comprehensive description of CSIP serves as a preamble to the detailed analysis of FeNS system.

3.3.1 Methodological approach

3.3.1.1 *NeuroEvolution of Augmenting Topologies (NEAT)*

Stanley and Miikkulainen in 2002 published the paper *"Evolving Neural Networks through Augmenting Topologies"*, describing a new evolutionary algorithm capable of gaining benefits from contemporary evolving both structure and weights of an artificial neural network (Stanley and Miikkulainen (2002)). At that time, several algorithms of the Topology and Weight Evolving Artificial Neural Networks (TWEANN)s family were able to simultaneously evolve both topology and weights already (Angeline et al. (1994); Braun and Weisbrod (1993); Dasgupta and McGregor (1992); Fullmer and Miikkulainen (1992); Gruau et al. (1996); Krishnan and Ciesielski (1994); Lee and Kim (1996); Maniezzo (1994); Opitz and Shavlik (1997); Pujol and Poli (1998); Yao and Liu (1998); Zhang and Muhlenbein (1993)). The open question in neuroevolution was about gaining advantage from the contemporaneous evolution of topology and weights. NEAT is designed to evolve the structure of a neural network starting from the minimal topology and incrementally growing it. This fasten the learning process by keeping the size of the search space of connection weights at its minimum. Stanley and Miikkulainen demonstrated how the best fixed-topology neural networks were outperformed by NEAT (Stanley and Miikkulainen (2002)). This paper was also a valuable contribution to research in genetic algorithms Genetic Algorithm (GA)s: NEAT algorithm, indeed, is able to progressively complexify and optimize the neural networks over generations Stanley and Miikkulainen (2002).

Solutions of four well established problems are the actual innovations behind NEAT algorithm:

1. TWEANN Encoding;
2. Competing conventions;

3. Protecting innovations;
4. Initial population and topological innovation.

The four issues are following briefly introduced. Subsequently, NEAT solutions are described.

3.3.1.1.1 Problems

Network encoding

There are two types of genetic representations for encoding an artificial neural network: direct and indirect encoding schemes.

Direct encoding is characterised by more explanatory representation. Every node and connection in the genome appears in the phenotype as well. For example, Structured Genetic Algorithm (sGA) Dasgupta and McGregor (1992) describe the connection matrix through the traditional bit string; Parallel Distributed Genetic Programming (PDGP) (Pujol and Poli (1998)) makes use of both a graph structure and a linear genome of node definitions to ease crossovering. Both structures show evident limits like a fixed number of nodes in the network, or number of nodes is consequence of human choice.

Indirect encoding has a less explicit representation because connections and nodes are explicitly defined in the genome but they can be extracted from it. For example, in Cellular Encoding (CE) (Gruau (1993)) specialized graph transformation language are utilized for programming the genomes in order to specify *cell division*. However, there wasn't a deep understanding of indirect encoding at that time, so one of major drawbacks was an uncontrolled way of searching for solutions. This led indirect encoding to focus on some suboptimal classes of topologies.

Competing conventions (permutations)

Competing conventions problems happen when:

A. **Several options are available to solve the weight optimization problem with neural network;**

Figure 9 illustrates two ANNs able to reproduce the same function with identical structure but different representation (chromosomes). Simply flipping hidden neuron 1 (H1) and hidden neuron 3 (H3) makes the two ANNs incompatible for crossover because there is a high probability of losing important information. The crossover of representation (H1,H2,H3) with (H3,H2,H1) can potentially result in (H1,H2,H1) or (H3,H2,H3), losing H3 information in the first case and H1 in the second one. Crossovering these genomes has high probability of resulting into damaged offspring.

B. **There is more than one topology to express the same neural network;**

C. **Genomes of different sizes can represent similar solutions.**

The crossover in cases B and C could potentially fail because neural network representations may not match up. In detail, this happens when genes are in the same position on different chromosomes but represent totally dissimilar traits.

Here, the constraint is positional crossovering: genes properly match up depending on their traits, even if their are located at different positions on different chromosomes. Actual alignment strategy is required.

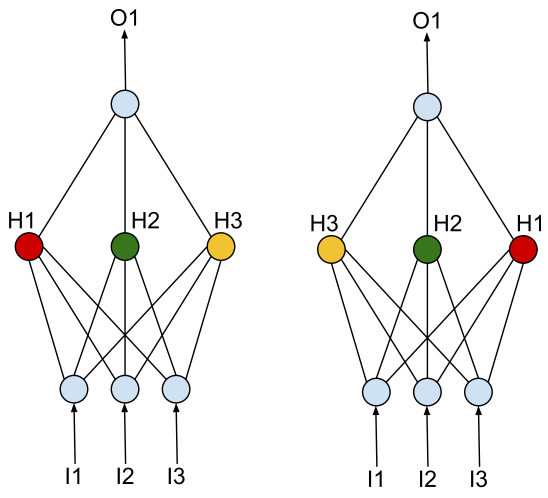


Figure 9: Competing conventions problem. The two ANNs have identical structure but different order of hidden neurons. Here, crossovering the two networks might result in missing one of the 3 hidden units.

Protecting innovation

The result of a mutation process generates a network with a new structure. Rarely the newly added node or connection are perfectly tuned and they usually don't have positive impact on network fitness. For example, when a new connection is added, its weight is likely to be not optimized consequently decreasing network performance. Equivalently, the addition of a new node to the network structure generates a new extra nonlinearity which becomes part of network behaviour. Furthermore, the addition of a new node automatically involves the addition of a new connection with a default weight value.

As a result, the new structure has to be optimized for some generation before its actual use. However, the loss of fitness penalizes the new network in the population which may not survive for enough generations to be properly optimized. This innovation has to be somehow protected long enough and actually check its goodness once maturity is reached.

Initial population and topological innovation

The initial population in TWEANNs systems is usually randomly generated which means that each genome starts with a random topology. This opens two further scenarios. A starting network may have:

A. no connection from each of its inputs to its output;

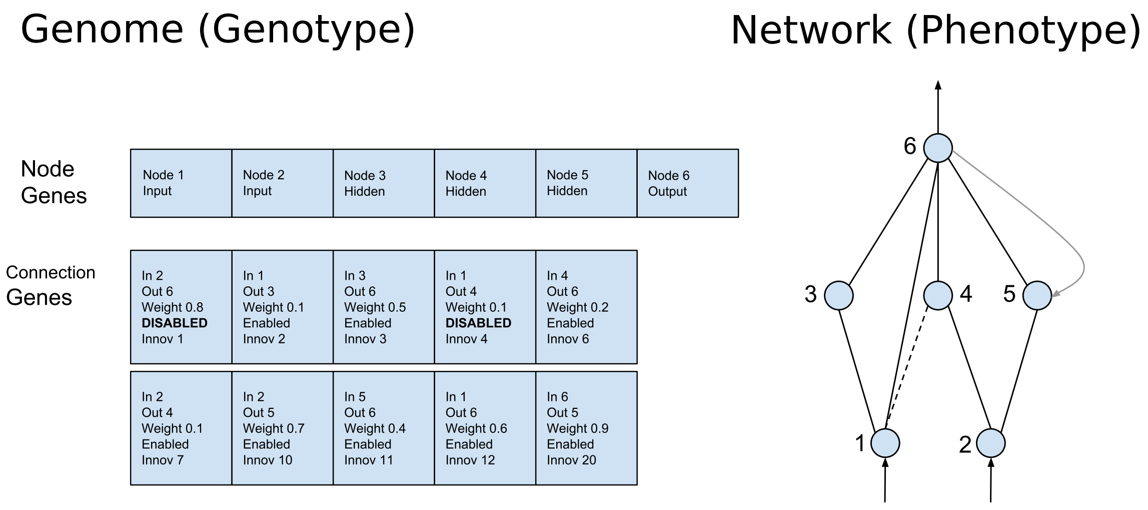
B. useless nodes or connections because they have never been evaluated before (and get rid of structures that shouldn't have to be there require additional avoidable efforts).

There shouldn't be any hidden node in the initial population in order to start with minimal topology networks. A structure should grow only if the final fitness has a positive impact.

3.3.1.1.2 Solutions

From direct to genetic encoding

NEAT makes use of direct encoding-type linear representation of network connectivity to describe genomes. Each genome contains two lists (see Figure 10):



Node 1 Input	Node 2 Input	Node 3 Hidden	Node 4 Hidden	Node 5 Hidden	Node 6 Output

In 2 Out 6 Weight 0.8 **DISABLED** Innov 1	In 1 Out 3 Weight 0.1 Enabled Innov 2	In 3 Out 6 Weight 0.5 Enabled Innov 3	In 1 Out 4 Weight 0.1 **DISABLED** Innov 4	In 4 Out 6 Weight 0.2 Enabled Innov 6
In 2 Out 4 Weight 0.1 Enabled Innov 7	In 2 Out 5 Weight 0.7 Enabled Innov 10	In 5 Out 6 Weight 0.4 Enabled Innov 11	In 1 Out 6 Weight 0.6 Enabled Innov 12	In 6 Out 5 Weight 0.9 Enabled Innov 20

Figure 10: The genome (left side) maps the actual network structure (right side). Two genes (node and connection) describe neuron types and their interconnections.

- **Node Genes:** available input, hidden, and output nodes;
- **Connection Genes:** each connection gene contains in-node, out-node, connection weight, if the connection is enabled and an *innovation number* (fully described in the next section, it allows for identifying same genes in different genomes during crossover).

NEAT evolves connection weights and network structure through mutation process. During each generation, connection weights get perturbed or not (standard NeuroEvolution (NE) connection mutation). Only two structural mutations are allowed:

- Add connection: this adds a new connection gene between two unconnected nodes with a random weight;
- Add node: this adds a new node between two connected nodes. Thus, the old weighted link is replaced by two new connections between the old nodes and the new one. This mutation strongly affects the current value of the fitness behaviour of the newly created neural network. To moderate this process, NEAT sets the weight of the connection to the new node to 1 while sets the weight of the connection from the new node identical to the old one.

In this way, the mutation lets the genome's size slowly increase.

Historical markings to overcome competing convetions problems

Natural dynamics are strongly inspirational for the development of GAs and the way they overcome competing conventions problem actually inspires NEAT solution as well.

Gene amplification (Darnell and Doolittle (1986); Watson et al. (1988)) is the process that allows for adding new genes to the genomes during sexual reproduction. Thus, even real genomes are not of fixed-length, otherwise there wouldn't have been any evolution from single cells to actual organisms. In order to successfully crossover, genes correctly align during the synapsis if they are homologous, i.e. they represent the same trait. This concept is named *homology*.

NEAT defines two genes homologous if they have same *historical origin*. Because they originate from the same ancestral gene, they represent the identical structure and match up for crossovering. This concept equates to nature's homology.

A *global innovation number* is introduced to keep track of the structural mutation chronological order. This index increments when a new gene is added and is assigned to it. For example, considering two subsequent structural mutations to the network in Figure 10: addition of node 5 (Figure 11), and addition of connection gene 1 - 6 contemporary to disablement of of connection gene node 1 - 4 (Figure 12). When node 5 is added, the two newly created connection genes between node 2 - 5 and node 5 - 6 get assigned the innovation numbers 10 and 11 respectively. If connection gene 1 - 4 gets pruned out immediately after, innovation number 12 gets assigned to newly created connection gene 1 - 6.

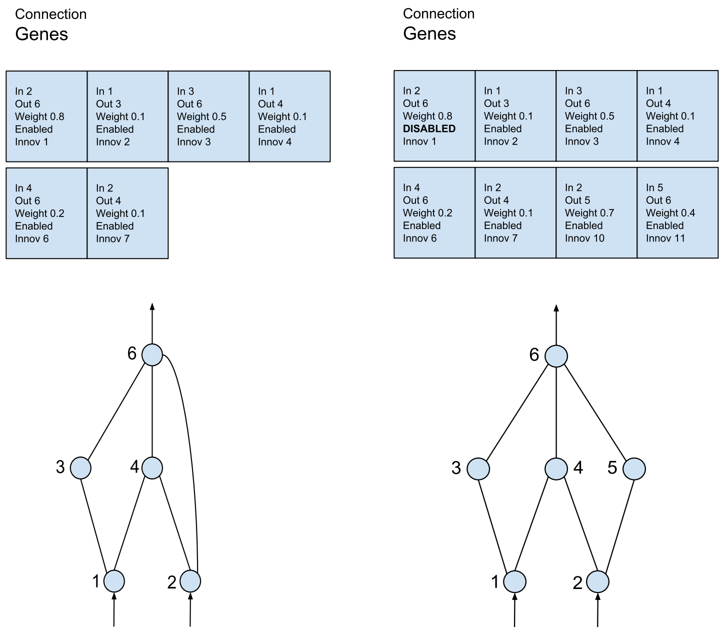

Connection Genes
In 2 Out 6 Weight 0.8 Enabled Innov 1
In 1 Out 3 Weight 0.1 Enabled Innov 2
In 3 Out 6 Weight 0.5 Enabled Innov 3
In 1 Out 4 Weight 0.1 Enabled Innov 4
In 4 Out 6 Weight 0.2 Enabled Innov 6
In 2 Out 4 Weight 0.1 Enabled Innov 7
6
3
4
1
2
Connection Genes
In 2 Out 6 Weight 0.8 DISABLED Innov 1
In 1 Out 3 Weight 0.1 Enabled Innov 2
In 3 Out 6 Weight 0.5 Enabled Innov 3
In 1 Out 4 Weight 0.1 Enabled Innov 4
In 4 Out 6 Weight 0.2 Enabled Innov 6
In 2 Out 4 Weight 0.1 Enabled Innov 7
In 2 Out 5 Weight 0.7 Enabled Innov 10
In 5 Out 6 Weight 0.4 Enabled Innov 11
6
3
4
5
1
2

Figure 11: Example of add node mutation. Left hand side illustrates the original ANN genotype and phenotype, while right hand side illustrates ANN genotype and phenotype after structural mutation.

When two genomes mate, genes with identical innovation number (*matching genes*) crossover and offsprings inherit the ancestor's innovation number. The choice between two matching is completely random. If two genomes have non-matching genes, *disjoint genes* (non-matching genes within the range of the other parent's innovation numbers) and *excess genes* (non-matching genes outside the range of the other parent's innovation numbers) are inherited from the more fit parent (see Figure 13).

Global innovation number is the only index required by NEAT to perform the artificial synapsis and properly line up matching genes. Additionally, NEAT collects the list of innovations that happened in the current epoch. Consequently, identical innovation number gets assigned to the same structural mutation that occurs more than once during the same generation.

Speciation to protect structural innovation

Nature developed the concept of niche to protect structural innovation. Different structures usually belong to different species. Challenges between species happen in different niches.

NEAT algorithm implements a similar concept: the entire population is divided into species, each species groups networks with comparable topologies so that

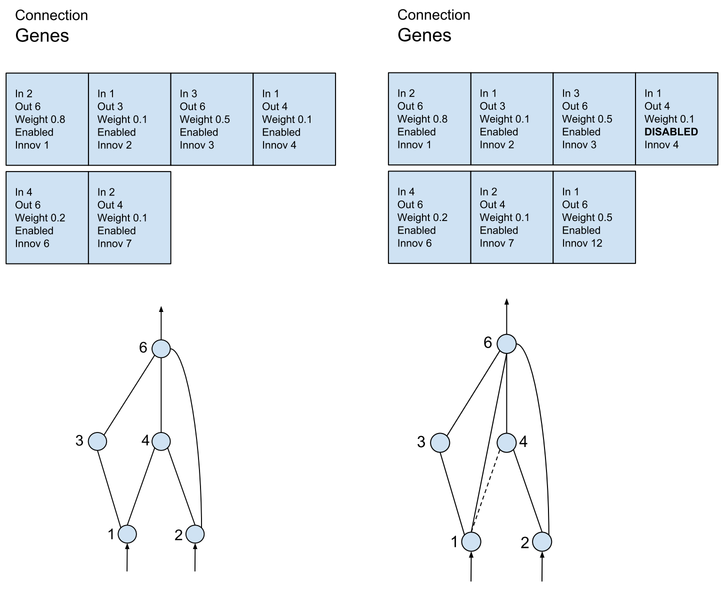

Connection Genes
In 2 Out 6 Weight 0.8 Enabled Innov 1
In 1 Out 3 Weight 0.1 Enabled Innov 2
In 3 Out 6 Weight 0.5 Enabled Innov 3
In 1 Out 4 Weight 0.1 Enabled Innov 4
In 4 Out 6 Weight 0.2 Enabled Innov 6
In 2 Out 4 Weight 0.1 Enabled Innov 7
Connection Genes
In 2 Out 6 Weight 0.8 Enabled Innov 1
In 1 Out 3 Weight 0.1 Enabled Innov 2
In 3 Out 6 Weight 0.5 Enabled Innov 3
In 1 Out 4 Weight 0.1 **DISABLED** Innov 4
In 4 Out 6 Weight 0.2 Enabled Innov 6
In 2 Out 4 Weight 0.1 Enabled Innov 7
In 1 Out 6 Weight 0.5 Enabled Innov 12
6
3
4
1
2
6
3
4
1
2

Figure 12: Example of add link mutation. Left hand side illustrates the original ANN genotype and phenotype, while right hand side illustrates ANN genotype and phenotype after structural mutation.

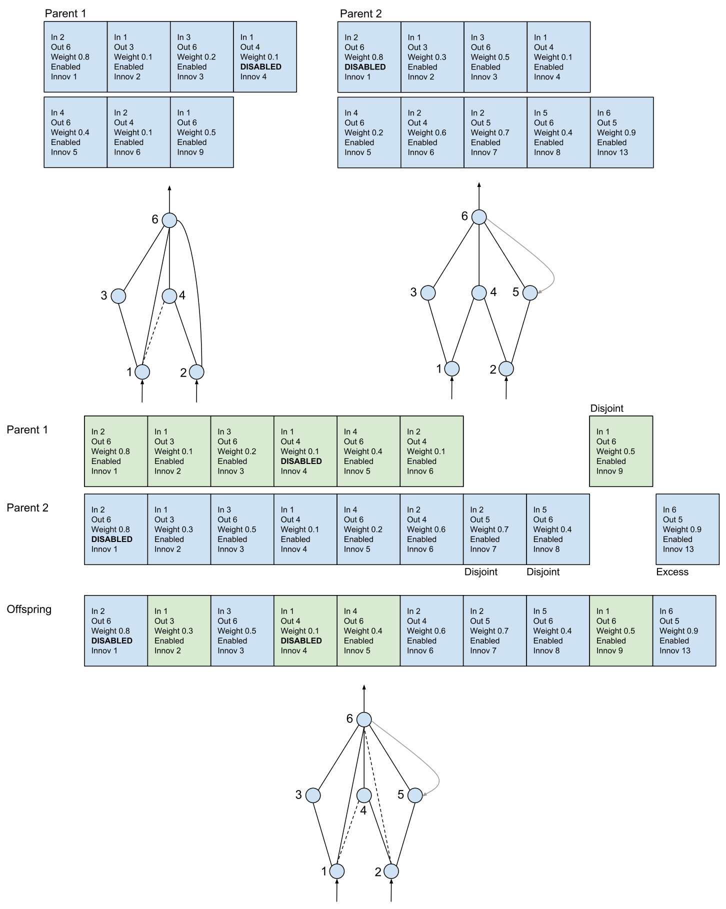

Parent 1
In 2 Out 6 Weight 0.8 Enabled Innov 1
In 1 Out 3 Weight 0.1 Enabled Innov 2
In 3 Out 6 Weight 0.2 Enabled Innov 3
In 1 Out 4 Weight 0.1 **DISABLED** Innov 4
In 4 Out 6 Weight 0.4 Enabled Innov 5
In 2 Out 4 Weight 0.1 Enabled Innov 6
In 1 Out 6 Weight 0.5 Enabled Innov 9
Parent 2
In 2 Out 6 Weight 0.8 **DISABLED** Innov 1
In 1 Out 3 Weight 0.3 Enabled Innov 2
In 3 Out 6 Weight 0.5 Enabled Innov 3
In 1 Out 4 Weight 0.1 Enabled Innov 4
In 4 Out 6 Weight 0.2 Enabled Innov 5
In 2 Out 4 Weight 0.6 Enabled Innov 6
In 2 Out 5 Weight 0.7 Enabled Innov 7
In 5 Out 6 Weight 0.4 Enabled Innov 8
In 6 Out 5 Weight 0.9 Enabled Innov 13
6
3
4
1
2
6
3
4
5
1
2
Parent 1
In 2 Out 6 Weight 0.8 Enabled Innov 1
In 1 Out 3 Weight 0.1 Enabled Innov 2
In 3 Out 6 Weight 0.2 Enabled Innov 3
In 1 Out 4 Weight 0.1 **DISABLED** Innov 4
In 4 Out 6 Weight 0.4 Enabled Innov 5
In 2 Out 4 Weight 0.1 Enabled Innov 6
Disjoint
In 1 Out 6 Weight 0.5 Enabled Innov 9
Parent 2
In 2 Out 6 Weight 0.8 **DISABLED** Innov 1
In 1 Out 3 Weight 0.3 Enabled Innov 2
In 3 Out 6 Weight 0.5 Enabled Innov 3
In 1 Out 4 Weight 0.1 Enabled Innov 4
In 4 Out 6 Weight 0.2 Enabled Innov 5
In 2 Out 4 Weight 0.6 Enabled Innov 6
In 2 Out 5 Weight 0.7 Enabled Innov 7
In 5 Out 6 Weight 0.4 Enabled Innov 8
In 6 Out 5 Weight 0.9 Enabled Innov 13
Disjoint
Disjoint
Excess
Offspring
In 2 Out 6 Weight 0.8 **DISABLED** Innov 1
In 1 Out 3 Weight 0.3 Enabled Innov 2
In 3 Out 6 Weight 0.5 Enabled Innov 3
In 1 Out 4 Weight 0.1 **DISABLED** Innov 4
In 4 Out 6 Weight 0.4 Enabled Innov 5
In 2 Out 4 Weight 0.6 Enabled Innov 6
In 2 Out 5 Weight 0.7 Enabled Innov 7
In 5 Out 6 Weight 0.4 Enabled Innov 8
In 1 Out 6 Weight 0.5 Enabled Innov 9
In 6 Out 5 Weight 0.9 Enabled Innov 13
6
3
4
5
1
2

Figure 13: Process of two mating parents.

when a structural innovation happens the new network fits into the more suitable niche where it optimizes and initially compete only with similar networks.

NEAT estimates a compatibility distance (delta) between networks in order to properly group them. Here, the solution to the competing convention problems comes to help.

> "The more disjoint two genomes are, the less evolutionary history they share, and thus the less compatible they are"
>
> – Stanley and Miikkulainen (2002) –

The distance between two genomes results from the linear combination of disjoint (D) and excess (E) genes, in addition to averaged weight differences of matching genes (W) (disable genes included):

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} \tag{3.1}$$

N is the number of genes in the bigger genome, c_1, c_2, and c_3 allows for balancing the significance of the three factors. δ is evaluated against a threshold δ_t.

In order to avoid a species to take over the entire population, NEAT reproduction mechanism is based off of *explicit fitness sharing* (Goldberg et al. (1987)). The final fitness (f') of each genome (i) is adjusted according to the distance (δ) from the other genomes (j):

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))}, \qquad \begin{cases} sh = 0 & \text{for } \delta > \delta_t \\ sh = 1 & \text{otherwise} \end{cases} \tag{3.2}$$

Thus, the final fitness is penalized by the number of genomes already presents in a specific species. Every species is assigned a potentially different number of offspring in proportion to the sum of final adjusted fitness of its member organisms. Worst performing genomes are eliminated during species reproduction, and surviving offspring in each species eventually replace the entire population.

Incremental growth from minimal structure

In the NEAT algorithm, population always start off with no hidden nodes. As a result, the search algorithm always looks for solutions in the minimal-dimensional space possible throughout all generations. In fact, a network, that starts from the minimal structure, grows only if the structural mutation improves solution quality.

This design clearly improve algorithm performance since the search space is always at its minimum.

3.3.1.2 *Feature selective NEAT*

FS-NEAT improves the standard NEAT algorithm by adding the *feature selection* process, which automatically selects inputs that yield the best ANN performance (Whiteson et al. (2005)). This process recognizes redundant and not significant parameters and prune them. This constraint the dimension of the search space to the lowest size possible.

FS-NEAT was developed for three main reasons (Whiteson et al. (2005)):

- alleviating human responsibility of properly setting up machine learning system by enabling the searching algorithm to select only the subset of the input parameters that increase learning performance;

- avoiding the presence of not relevant inputs which can slow down or even impede learning;
- controlling the pruning out of useless input parameters which can be computationally expensive. This leads also to ask as less features as possible to the end-user willing to apply the trained ANN.

Before FS-NEAT, feature selection methodologies were grouped in two categories (Langley (1994)):

A. **Filters** (Bonnlander and Weigend (1994); Kira and Rendell (1992)). These methodologies are based off of the analysis of labeled data. Filters don't account for the actual learning algorithm that will use the data.

B. **Wrappers** (Narendra and Fukunaga (1977); Pudil et al. (1994)). These methodologies make use of a meta-learner to evaluate the feature sets which is computationally expensive.

FS-NEAT overcomes previously listed constraints by selecting the the most suitable features during the learning task. FS-NEAT differs from regular NEAT only for the setup of the initial population. NEAT algorithm generates a population where each input of a neural network is connected to each output (Stanley and Miikkulainen (2002)). This surely generates small networks to start with. However, those are not the smallest networks possible. Additionally, with this kind of structure the learning algorithm assumes that every input is useful to gain the best performance. Which may not always be the case (Whiteson et al. (2005)).

The smallest dimensional space to start with would be a pool of completely disconnected input and output nodes with no hidden nodes. However, this kind of network requires at least one generation to start producing outputs, which would be prodigal (Whiteson et al. (2005)).

FS-NEAT initializes the population in the following way (see Figure 14):

1. each output node is linked to a bias node;
2. considering a set of I inputs, each node $i \in I$ has a probability $\frac{1}{|I|}$ to get linked to every output node.

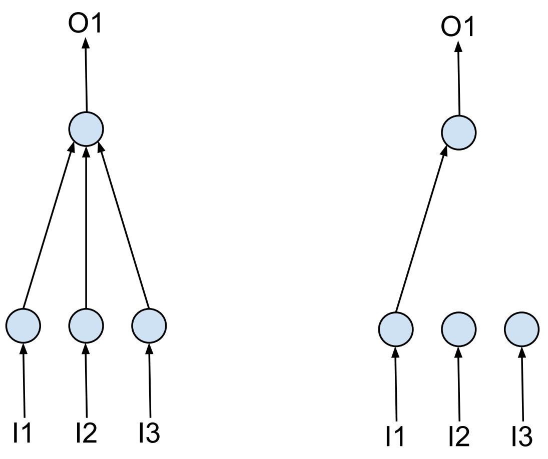


Figure 14: Left hand side shows the initial structure of NEAT generated ANN. Right hand side shows the initial structure of a FS-NEAT generated ANN.

This strategy allows for creating a very diversified initial population. Then, the evolutionary algorithm will select the most performant individuals and start applying structural mutations and crossover.

3.3.1.3 *Ensemble of surrogate models and uncertainty quantification*

The ensemble of surrogate models approach (Serafin et al. (2018b)) results from combining the stochasticity inherent to the evolutionary genetic algorithm implemented in NEAT to a specifically designed cross validation-like technique.

NEAT evolves connection weights and network structure through mutation process (Stanley and Miikkulainen (2002)). Accordingly, the neuroevolutionary algorithm involves two type of stochastic mutation processes:

1. connection weight perturbation, which may or may not happen depending upon standard neuroevolution connection mutation;
2. structural mutation, which involves random addition of a connection gene between two disconnected nodes, or a node gene between two already connected node genes.

This stochastic procedure already generates a slightly different ANN every time a training process fed with identical input/output snapshots starts. Training an ensemble of ANNs results in several SMs capable of properly emulating the original model behaviour even if they differ in structural topology and connection weights.
Instead of choosing one single ANN out of a trained ensemble, an accurately designed cross validation-like procedure is used to emphasize this behavioural stochasticity.
Supervised learning methodologies require to split the available dataset into three sets: training, validation, and testing (Govindaraju and Rao (2000a); Friedman et al. (2001); James et al. (2013)). Thus, the entire dataset is initially divided into two groups: training+validation (TV) dataset, and testing (see Figure 15).

Train
Validation
Test

Figure 15: Dataset splitting in training, validation, and testing.

Afterwards, right before the beginning of the learning process, the TV dataset is randomly split into training and validation datasets. The partitioning algorithm makes sure that the two sets have same probability distribution, in order to break down the TV dataset into two significant samples.
As a result of this splitting procedure, each ANN is trained and initially validated against slightly different datasets, ending up with a unique structure capable of emulating original model behaviour.

The ensemble system results from concurrently training several ANNs and following selecting the utmost performant ones. The ensemble system is then collectively run to provide uncertainty quantified results against the testing dataset and subsequently applied against new data.

3.3.1.4 *Framework-enabled NEAT based Surrogate modeling (FeNS)*

Framework-enabled NEAT based Surrogate modeling (FeNS) concept has been designed to support, facilitate, and automate the transitioning from research models to *"field models"*.

Model complexity in terms of data resolution requirements, number of parameters, platform affinity, calibration or sensitivity analysis procedures, causes problems in consultant environments for timely delivery of research models themselves, IT deployment infrastructure management, model usability and data provisioning for on-the-field personnel, performance expectations, and field user training (David et al. (2013)).

The use of EMFs and CCPs such as CSIP/OMS (David et al. (2013, 2014a); Lloyd et al. (2012)) alleviated some of the implications for model users (David et al. (2012); Lloyd et al. (2011)). The introduction of these software in the modeling workflow is represented by Step 1 in Figure 16.

Independently from model type, size, and complexity, any research simulation model is encapsulated into model components or web-services. As a result, user-model interface is simplified and unified by the adoption of standard input/output data formats (approach complexity in Figure 16). Legacy models are effortlessly encapsulated as black-box software applications into EMFs/CCPs workflow as well.

Eventually, EMFs/CCPs moderate run-time issue by leveraging innovative hardware infrastructure such as super-computing environments and computer clusters (David et al. (2013, 2014a); Lloyd et al. (2012)).

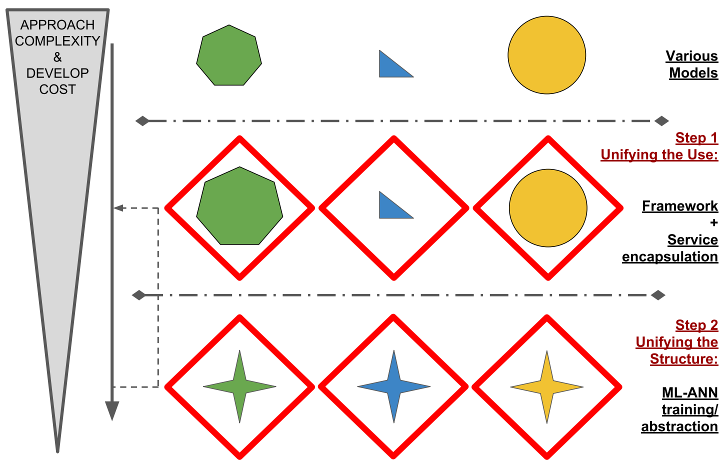

APPROACH COMPLEXITY & DEVELOP COST
Various Models
Step 1 Unifying the Use:
Framework + Service encapsulation
Step 2 Unifying the Structure:
ML-ANN training/ abstraction

Figure 16: With respect to Figure 2, this schematic illustrates the introduction of a second step. In addition to framework encapsulation, original research model runs generate SMs, which can be used with little or no user effort and don't require any model maintenance and development cost.

Nevertheless, EMFs/CCPs don't completely overcome service delivery organization issues with respect to operational use of mathematical models. Calibration and sensitivity analysis procedures, and high resolution data are still required for a model simulation run, which turns out to be computationally demanding nonetheless.

FeNS attempts to overcome actual constraints by providing service delivery organizations with an easy to use, lightweight, automatically generated tool which derives from original simulation model.

In addition to unification of user-model interface (Figure 16 - Step 1), this tool, namely ANN-based surrogate model, allows for inner structure unification (Figure 16 - Step 2): here, SMs originating from different conceptual/physical models shows identical structure, which is composed by a variable number of functional units (nodes and connections) and is completely decoupled from original simulation model internal structure. This allows for applying the SM as a black-box without any specific modeling proficiency.

Additionally, by training SMs to emulate original model behaviour using relevant input entries only, user-approach complexity is minimized.

In conclusion, the SM allows for emulating original simulation model by:

- reducing the number of input parameters required;
- answering application specific questions almost instantaneously without asking for any modeling proficiency;
- minimizing SM development and deployment cost, automatically generating the SM at a framework level (Figure 16 – Step 2)

3.3.1.4.1 FeNS concept

EMFs/CCPs foster research simulation model runs and directly manage input/output dataset.

Since each input/output model run implicitly represents part of the original model behaviour, that piece of information is of fundamental importance to emerge a properly calibrated SM. Consequently, EMFs/CCPs need to be utilized to harvest input/output model runs and emerge SM at the modeling framework level.

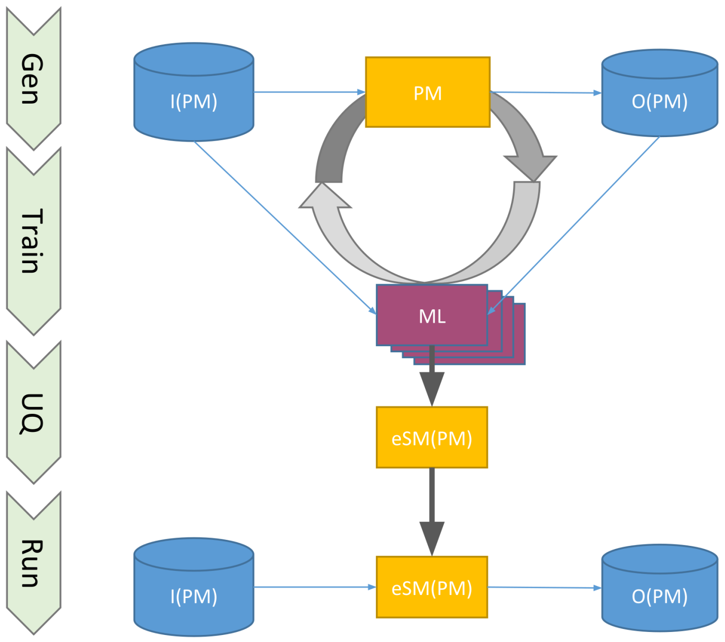


Figure 17: Generic FeNS concept. ML library employes original model runs to emerge eSM from any modeling solution.

Figure 17 explains the concept:

1. EMFs/CCPs-compliant conceptual/physical model (PM) runs generate (Gen) model outputs (O(PM)) corresponding to provided input data (I(PM));
2. EMF/CCP harvests I(PM)/O(PM) data snapshots;
3. when a sufficient amount of I(PM)/O(PM) snapshots are collected, NEAT neuroevolutionary ML algorithm (ML) starts emerging (Train) n-SMs;
4. once n-SMs are created, the utmost performant ones are selected to compose (UQ) the ensemble of surrogate models (eSM) (eSM(PM));
5. finally, the eSM(PM) is queried by the end user to accurately answer application specific question.

3.3.1.4.2 FeNS protocol

FeNS protocol defines a set of rules for integrating FeNS approach into framework workflow requiring little or no user effort.
To begin with, FeNS concept is split into two main stages:

STAGE A. harvesting of input/output snapshots of original model runs;

STAGE B. automated emerging of eSM from any modeling simulation.

Here, STAGE A requires intervention on current model simulation workflow and hence is the only stage that affects model user experience. STAGE B might require some dedicated SM input parameters set up but mostly necessitates of designing ad-hoc applications for asynchronously emerging the eSM.

FeNS protocol regulates issues and constraints related to integration of both stages in framework workflow. Problem decomposition into elementary units allows for identifying and analyzing required rules.

STAGE A: harvesting of I/O snapshots of original model runs

This is the only stage that affects model user experience since it involves original model simulation workflow.

While interfacing with a model run, user is required to identify (or tag) relevant model inputs to properly describe selected model outputs. This tagging system allows FeNS to recognize the parameters to harvest from the original model run and collect afterwords.

In addition to tagging parameters available from within model configuration file, model user might want to perform mathematical operations on specific input parameters to reduce input space dimensionality. This might involve model parameters retrieved from thirty parties data providers such as remote databases or other model dependencies. Consequently, each original model needs to expose data provider dependencies.

FeNS identifies two model simulation workflow aspects which need to accommodate model parameter tagging and expose model data dependencies:

A. **user-model interface** to extend tagging of input parameters;

B. **simulation model source code** to expose model data dependencies.

The next two paragraphs introduce to the set of non-invasive changes required to integrate FeNS methodology in the framework-compliant model workflow.

For the sake of example, the description of this protocol is based off of OMS/CSIP EMF/CCP input/output standard format (JavaScript Object Notation (JSON) payload), and RESTful compliant CSIP-services.

A. USER-MODEL INTERFACE A generic model user interfaces to the simulation model through a configuration file such as JSON payload. Here, user lists required model input entries and input settings.

To trigger the SM generation process, user needs to select relevant model input parameters, identify model results, and potentially perform mathematical operations to aggregate array or maps of input data.
FeNS identifies 5 categories of input/output parameters:

- STANDARD input parameters;
- COMPUTED input parameters;

- DEPENDENCY DERIVED input parameters;
- ADDITIONAL input parameters;
- OUTPUT model parameters.

Input model parameters are described through JSON Objects in a standard JSON payload and adhere to the formal structure in Listing 3.1.

Listing 3.1: JSON Object of a generic model input parameter.

```
{
    "name": "param_name",
    "value": "param_value"
}
```

FeNS introduces the additional JSON key "description" which allows a user to:

1. identify which input/output parameter has to be collected: "collect_in" or "collect_out" respectively;
2. describe the mathematical operation to perform: "math_expression" (e.g. averaged sum) or "methodology" (e.g. kriging);
3. list the dependency to derive the parameter from (this element might not be mandatory): "dependency" (e.g. database name).

Modified JSON Object of each category of model parameters is following described.

A.1 STANDARD input parameter No operation needs to be performed on a standard input parameter, which simply has to be tagged and collected by FeNS system, consequently (see Listing 3.2)

Listing 3.2: JSON Object for STANDARD input parameter.

```
{
    "name": "param_name",
    "value": "param_value",
    "description": "collect_in"
}
```

A.2 COMPUTED input parameter This input parameter might require mathematical operations to be reduced to a single relevant value. This parameter might be in the form of array, map, etc. Consequently, the description field needs to contain the mathematical operation to perform to the input parameter. This operation might be:

- mathematical expression;
- methodology such as lumped kriging on watershed centroid.

JSON formal structure of this input parameter might look like Listing 3.3.

Listing 3.3: JSON Object for COMPUTED input parameter.

```
{
    "name": "param_name",
    "value": [val0, val1, ...],
    "description": "collect_in,math_expression/methodology"
}
```

A.3 ADDITIONAL input parameter This parameter is not part of standard model input dataset. However, it is derived as a result of mathematical operations performed on a list of actual input parameters. This allows for applying input dimensionality reduction and summarizing the behaviour of several input parameters into a single one. JSON formal structure resemble Listing 3.3 but doesn't contain any explicit value (see Listing 3.4).

Listing 3.4: JSON Object for ADDITIONAL input parameter.

```
{
    "name": "param_name",
    "description": "collect_in,math_expression/methodology"
}
```

A.4 DEPENDENCY DERIVED input parameter This parameter is not available in the original input set of model parameters and is actually derived from data provider model dependencies. Here, the dependency to look for might be specified in the JSON payload (see Listing 3.5).

Listing 3.5: JSON Object for DEPENDENCY DERIVED input parameter.

```
{
    "name": "param_name",
    "description": "collect_in,
                    math_expression/methodology,dependency"
}
```

A.5 OUTPUT parameter The output parameter is not obviously part of the original model JSON payload. However, it has to be identified and collected (see Listing 3.6).

Listing 3.6: JSON Object for OUTPUT parameter.

```
{
    "name": "param_name",
    "description": "collect_out"
}
```

B. SIMULATION MODEL SOURCE CODE The simulation model is not asked to perform any additional operation on the parameter set. The simulation model needs to collect data provider dependencies information only (e.g. database connections) and returns them along with model results. Still using JSON payload for the sake of example, this informations are collected into the JSON response.

A simulation model is required to implement one additional method such as `putDependencies` (Listing 3.7) provided by the framework API.

Listing 3.7: Example of a `putDependencies` method.

```
@Override
public void doProcess() throws Exception {

    ...

    putDependencies(<list of dependencies>);

}
```

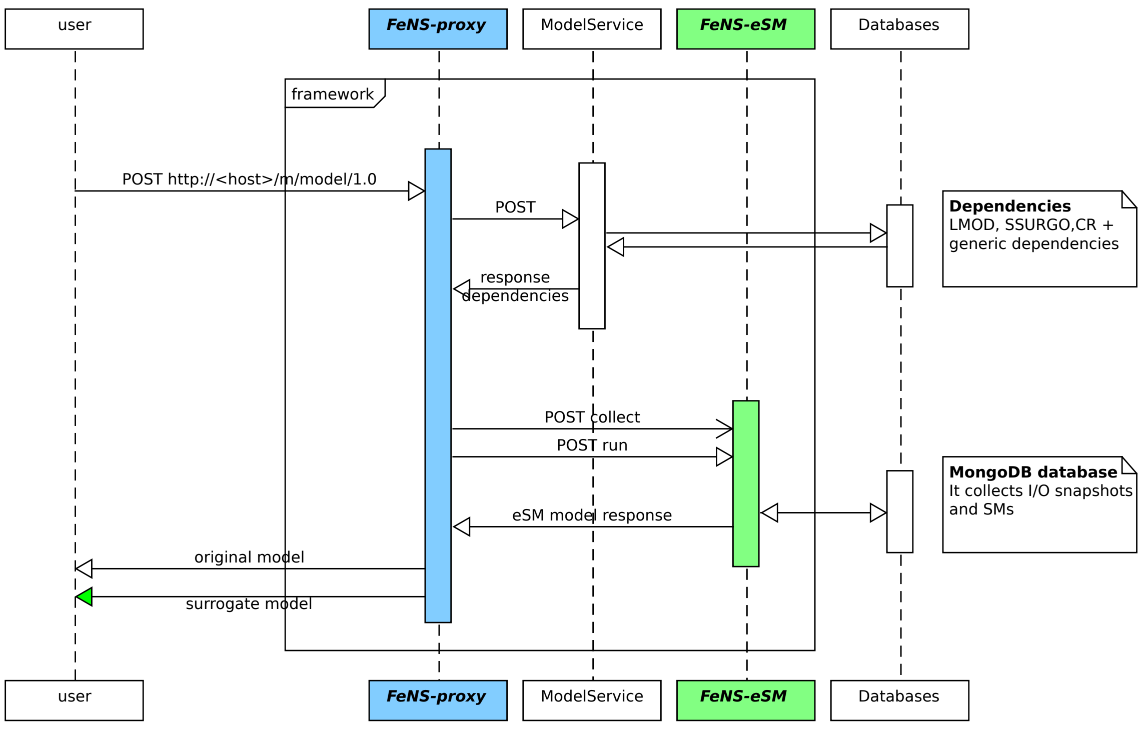


Figure 18: FeNS architectural design. FeNS-proxy interposes between user and model service, orchestrates parsing, retrieval and computing input/output original model parameters, and finally triggers surrogate model generation by feeding FeNS-eSM with input/output snapshots.

STAGE B: automated emerging of eSM

The automated creation of the eSM relies on ad-hoc development of framework applications against ML libraries.
To emerge the eSM, FeNS system identifies 4 steps:

1. data collection into dedicated database;
2. data normalization;
3. SM training (actual SM creation);
4. SM run against new dataset.

Specifically, last step replaces original model run when SM estimates become accurate enough. To provide this functionality, a further application might be required to select the most performant SMs only. This application is located between SM training (step 3) and SM run (step 4) and it is identified as SM selection. FeNS set of applications is following referred to FeNS-eSM.

3.3.1.4.3 FeNS architectural design

FeNS protocol identifies the set of rules for collecting model parameters and the main steps for automatically emerging the SM at a framework level.
FeNS architectural design defines the architectural aspects that allow for actually accomplish the goal of SM creation by subjecting to FeNS protocol constraints.
Figure 18 illustrates FeNS architectural design.
FeNS system relies on the introduction of two additional elements to the standard framework workflow:

A. FeNS-proxy

B. FeNS-eSM

A. FeNS-proxy

FeNS-proxy orchestrate the actual connection between original model service and FeNS-eSM to trigger surrogate model generation. This allows for moving the burden of collecting/processing input and output data aside from user or original model service, thus avoid original model modifications.

FeNS-proxy is in charge of parsing the input JSON payload and identifying the input parameters already available for collection. It additionally provides computational capabilities for operating mathematical equations to aggregate input array or maps or deriving additional parameters.

FeNS-proxy synchronously waits for model service to return output JSON response and list of data providers to fetch additional information from. When required data are retrieved from data providers, FeNS-proxy terminates pending computations and packages input/output snapshot for triggering SM generation.

Eventually, when the eSM is completely generated, FeNS-proxy looks up to a list of eSM available and queries the surrogate model instead of the original model.

B. FeNS-eSM

FeNS-eSM is a set of microservices that interacts with FeNS-proxy and MongoDB database to generate the ensemble of surrogate models. These services orchestrate the operations identified by FeNS protocol STAGE B:

1. Data collection;
2. Data normalization;
3. Surrogate model training;
4. Selection of most performant surrogate models;
5. Run of the eSM against new data.

Section 3.3.2.5 *Surrogate Model Services implementation* at page 55 exhaustively describes the current implementation of FeNS-eSM CSIP-services.

3.3.2 Technical approach and implementation

3.3.2.1 *MongoDB*

MongoDB is a NoSQL database that arose in the early 2000s. It was created by Dwight Merriman and Eliot Horowitz to overcome limitations of standard SQL databases when dealing with big data.

MongoDB *"doesn't try to be everything to everyone"* (Chodorow (2013)). It rather tackles the two principal issues a db developer faces when working with increasing amount of data and web applications: speed and scalability.

For example, it's common practice to initially set up a single server to store limited amount of data. When the volume of data starts passing the threshold of terabyte in size, the developer needs to design a replication set up to properly scale out reads, a caching layer coupled with fine queries tuning to reduce db response time, data sharding to accurately spread out data across multiple machines. Eventually, the developer ends up redesigning the entire database because the schema initially chosen locks any kind of development and expansion (Chodorow (2013)). This lack of flexibility due to the use of fixed schemas, tables, and rows is the real constraint of relational SQL databases. By contrast, MongoDB is built on the notion of collections and documents. Documents are sets of key-value pairs and are the fundamental data unit. They have no predefined or fixed schema (nested documents are allowed) which allows for easing the experimenting process in order to identify the best set up, and document modifications because fields can be added or removed with no type or size constraints (Chodorow (2013)). Collections contain sets of documents with different field structures. For the sake of analogy, documents and collections are like rows and tables in relational databases language.

Document is the core concept that allows MongoDB databases to be massively scalable and sharded. Data in documents are self-contained and key-values pairs structure facilitates queries. Thus, documents can easily split up across different machines which are in charge of updating their own subset of the entire dataset (data sharding) (Plugge et al. (2015)). Furthermore, these features well fit active/active cluster configuration: two or more actively running nodes concurrently check if they store data required by a specific MongoDB query/request; the only server containing desired information responds to the request (Plugge et al. (2015)). This is the most optimized configuration in terms of load balancing. The workload is accurately redistributed across the cluster, preventing overload issues in a single node.

MongoDB capabilities are elevated by this scale out configuration (horizontal scalability across multiple nodes) rather than a scale up one (vertical scalability on a bigger machine). Vertical scalability is typical of relational databases which are not suitable for active/active cluster configuration. It is surely the easiest set up but a bigger and faster machine is expensive and no more improvements are possible when physical limit is reached (Chodorow (2013); Plugge et al. (2015)). On the other side, horizontal scalability is cheaper because relies on several smaller connected machines but it is hardly manageable. Here MongoDB plays a key role because it automatically takes care of balancing data and load across available machines, redistributing documents and figuring out how to spread data when a new machine is added to the cluster (Chodorow (2013)).

In terms of performances, MongoDB workload is based off of maximizing random-access memory (RAM) usage by caching queries indexes and further queries informations, and by preallocating data files. Furthermore, MongoDB stores documents in Binary jSON (BSON) format, which is binary form JSON format. The latter is perfectly suitable for exchanging and storing data in a self-contained schemaless document (Plugge et al. (2015)). MongoDB uses BSON instead of plain JSON because it speeds up processing and searching operations through stored files and allows for managing binary data.

MongoDB provides *geospatial indexing* as well, indexing technique that allows for selecting location-based data, e.g. querying stored data within a given range (Plugge et al. (2015)).

In order to purse speed and massive scalability, MongoDB developers chose not to include *transactional semantic*[1], *joins* and *master/master replication*[2] features in the architectural design. However, if transactional support is required, a hybrid configuration of SQL and NoSQL databases is the best solution. This allows for contemporary leveraging the most suitable features of both type of databases.

1 Transactional semantic is a feature of SQL databases which guarantees data consistency even during power failures or software errors. It satisfies the ACID (Atomicity, Consistency, Isolation, Durability) properties.

2 Master/master replication (or multi-master replication) concept allows different servers to accept write requests.

3.3.2.2 *Microservice architecture and RESTful API*

Cloud computing systems are state-of-art in terms of modeling environments. In order to leverage available hardwares, cloud computing systems rely on microservice architecture as Service Oriented Architecture (SOA), and REpresentational State Transfer (REST)ful API as the most widely used way to interface users and remote services. This section introduces to the concepts of microservice/microservice architecture and REST.

Microservice Architecture (MA) was developed to overcome issues and constraints related to deployment and resiliency of large monolithic applications (Nadareishvili et al. (2016)). The main goal was to break up big applications into several small services (*services granularity*) in order to build a system of replaceable over maintainable piece of software.

Nadareishvili et al. in *"Microservice architecture: aligning principles, practices, and culture"* (2016) defines a microservice as

> "[...] an independently deployable component of bounded scope that supports interoperability through message-based communication,"

and a microservice architecture as

> "[...] a style of engineering highly automated, evolvable software system made up of capability-aligned microservices."

There are no strict definitions of microservice and microservice architecture. They rather change from company to company because it mostly comes to achieving three goals: speed and safety at scale (Nadareishvili et al. (2016); Newman (2015)). Speed relates to the need of quickly change and deploy a specific part of an entire application. Safety is always a fundamental aspect: no matter how quick you make changes and redeploy the microservice, if you break the production system your efforts are useless. Thus, the proper trade-off between speed and safety is a basic element. Scale links to the inevitable aspect of growing demand to access an application. Thus, software has to be designed and built to deal with demand that can grows over beginning expectations (Nadareishvili et al. (2016)).
To achieve these goals, every microservice application has to be (Nadareishvili et al. (2016); Newman (2015)):

- **Small size:** there can't be a single size as a generic rule for every microservice. There is the *cohesion principle* , which is a stronger concept of *single responsibility* (Martin (2009)). Basically, a microservice gathers every single aspect of a bigger application that changes for the same reason.
- **Bounded by contexts:** a microservice needs to be self-contained within boundaries of a more complex system. This allows for increasing cohesion of each single module, reducing coupling between modules in the system. Furthermore, this philosophy opposes to *one-size-fits-all* approach and rather facilitates polyglotism of technologies (graph-oriented database for describing the international relationships of a company vs document-oriented database for collecting communications between companies), languages and frameworks. This aspect allows for developing every microservice with the most suitable programming language, shorten the process of software development.
- **Independently deployable:** when a microservice is small and bounded by contexts, it is also independently deployable. Software developer can modify or reimplement one single service and deploy it. There is no need for large

and long deployment anymore. Furthermore, this allows for fast rollback if the developed component of the entire system fails. The microservice architecture avoid a failure cascade. The production environments maintains its working status, while MA gracefully degrades system functionalities avoiding a total failure.

- **Messaging enabled:** communications to and between microservices happen by posting requests through the net. This configuration enables the *systematization*. If MA eases the process of microservices design, burden and complexity are handled by the system. The latter is designed to manage modules communication. It doesn't deal with internals and behaviours of each module. This design eases the process of handling growable systems.

- **Autonomously developed:** when dealing with big monolithic applications, none of the software developers in the team knows everything about that software. In case of failure, bug fixing is a slow process because software developer has to go through the entire code and find out the cause of the failure. Same procedure happens when it comes to maintain or modify the monolithic application. MA allows for setting up autonomous and context experts teams of developers. Microservice cohesion reduces dependencies between teams. As a results, the process of making code available to production is faster and independent from each other module.

- **Disposable:** when a microservice is not updated and not suited for a specific application, it can be easily disposed and a new microservice redesigned. The set up of *autonomously developing* MA facilitates the design of *disposable code*.

- **Decentralized:** if there is no central body managing the entire system, there are fewer bottlenecks in the process to make developed code available in production environment. The bulk of work is spread out between independent teams of developers who speed up development/deployment operations.

- **Built and release with automated processes:** continuous integration (CI) automates the building and testing process by adding an automated layer of safety where every test runs constantly. This reduces the probability of deploying software with errors. This software development practice effectively impacts production code quality only if Unit Tests covering each and every MA capability are developed accordingly. The Test-Driven Development (TDD) movement sponsors this policy through important practical laws (from Martin (2009, 2007)):

  > **First Law:** You may not write production code until you have written a failing unit test.
  >
  > **Second Law:** You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
  >
  > **Third Law:** You may not write more production code than is sufficient to pass the currently failing test.

  To sum up, production code and unit tests are tightly developed simultaneously. As a result, software improvements that break actual state of the computational system are caught and identified immediately.

REpresentational State Transfer (REST) is an architectural style introduced by Fielding and Taylor in his PhD dissertation *"Architectural Styles and the Design of Network-based Software Architectures"* (Fielding and Taylor (2000)). Fielding's goal is to regulate communication between client and server by defining a set of software engineering principles. Detailed requirements of web services (component) implementation and descriptions of protocol syntax are not part of REST constraints. The latter rather focus on roles of components, their interactions, and data elements interpretation (Fielding and Taylor (2000)). REST-compliant web services are called RESTful web services, while their exposed APIs are RESTful APIs.

The actual REST-compliant architecture is the Resource-Oriented Architecture (ROA), introduced by Richardson and Ruby in *"RESTful web services"* (Richardson and Ruby (2008)). In every RESTful system there are three main actors involved:

1. **Uniform Resource Identifier (URI):** it is a label that univocally identifies and addresses a resource (Allamaraju (2010)). It is the name of that resource. Resource doesn't have a concrete definition. A resource is identifiable with any concrete or abstract object, representable with a binary sequence, storable on memory drives, and identified by a URI (Richardson and Ruby (2008)).

2. **HyperText Transfer Protocol (HTTP):** it is the protocol that provides the methods for defining the *uniform interface*. This allows the user to interact with web resources by sending requests and getting response through a uniform and predefined set of verbs. These are GET, POST, PUT, and DELETE.

3. **Representation (XML/JSON/HTML):** it is the data or metadata (an e-book and its reviews or cover image respectively) of the actual state of the resource (Richardson and Ruby (2008)).

A RESTful system follows the 6 constraints defined by Roy Fielding. If any of the required principles is violated, that system cannot be defined as RESTful.

1. CLIENT-SERVER ARCHITECTURE: this allows for separating concerns between the actual user interface exposed to the client and the data stored in the server. The benefit client-side is a wider portability of the user interface which can be designed to adhere to different platforms requirements. The benefit server-side is an improved services granularity which eases scalability, service simplicity and independent development and deployment;

2. STATELESSNESS: there is just one interaction between client and server. The client submits a request which makes him transitioning to a new state. This transitioning state lasts until the server returns a representation (the response), which contains every link to allow the client to start a new state transition. In this simple communication process, the server doesn't store any client context while the client maintains the session state instead. This principle improves *reliability* because it eases the process of recovering from partial failure; *scalability* because the server doesn't store any information from the client in between requests allowing for freeing resources at the end of each request and processing parallel interactions; *visibility* because the client request contains every information required by the service to correctly process the response, no monitoring system is required to look for previous client requests data in order to fulfill the current request;

3. CACHEABILITY: server responses implicitly or explicitly state whether they are cacheable or not. This allows clients or intermediaries to cache only suitable data preventing from caching stale or not appropriate data. Data caching reduces latency and improves user-perceived performance consequently. The user interacts with client cache or an intermediary, limiting client-server communication and improving services scalability;

4. LAYERED SYSTEM: this is a system with one or more layers (intermediary servers) in between client and server. The client may not communicate with the end server but he gets identical resources. As a result, load balancing is effectively managed, caching is provided on different levels, and some layers might be responsible of enforcing security policies. The overall system has a better scalability but overhead and latency could be downside of data processing;

5. CODE ON DEMAND (OPTIONAL): a server can send executable code to the client to extend functionalities during the session;

6. UNIFORM INTERFACE: this concept is the key of REST design. It allows for standardizing the client-server interaction, improving decoupling and facilitating independent evolution of each part involved consequently. Four constraints separate REST from other architectural styles:

   a) *Resource identification in requests:* a resource identifier URI is used to identify a specific resource in the request sent from a component to the other.

   b) *Resource manipulation through representations:* once the client receives a resource representation and related metadata, he owns every information required to alterate or delete that resource.

   c) *Self-descriptive messages:* every message involved in the client-server communication contains all the information the connector (client, server, cache, resolver, tunnel) needs to properly process it, e.g. each media type (data format of a representation) indicates which parser the connector has to use.

   d) *Hypermedia as the engine of application state (HATEOAS):* each server response to a client request has to contains all the hyperlinks that allow the client to dynamically keep changing state by looking through further available actions and resources.

The drawback of a uniform interface comes to efficiency degradation. A standardized form is used to transfer information indeed, which cannot be as efficient as a optimized one.

3.3.2.3 *Cloud Service Integration Platform (CSIP)*

The Cloud Service Integration Platform CSIP (David et al. (2014a)) is a Model as a Service (MaaS) (Zou et al. (2012)) platform that provides access to research modeling simulations as cloud based web-services. The CSIP hosting environment is designed for (i) elevating computational scalability, (ii) facilitating software application modularity, (iii) improving cost-productivity trade-off, while (iv) providing and open deployment system.

A MaaS platform resembles the Software as a Service (SaaS) platform behaviour: it enables highly granular software applications reusability over the network but it narrows down its scope to (environmental) models rather than generic software components (Roman et al. (2009)).

A MaaS platform works as a service provider, thus simplifying model user access to research modeling solutions. As a matter of fact, in addition to move the burden of managing execution environments from model users to hosting environment, it can potentially satisfy model input data requirement automatically (David et al. (2014a)).

A MaaS is built on top of a CCP, which is a pool of virtual computational resources accessed over the network. These resources (hardware, platforms, and web-services) are allocated and disposed on user's request. Advantages of CCPs vary from (1) no in-house resources maintenance, development and administration; (2) secure access and data protection; (3) guaranteed availability; (4) flexible updates of operating systems and applications on a large scale (David et al. (2014a)).

Only recently, this computational environment became appealing to the scientific community, which discovered its potential and capabilities and started make a viable option for research projects (Jha et al. (2011); David et al. (2014a)).

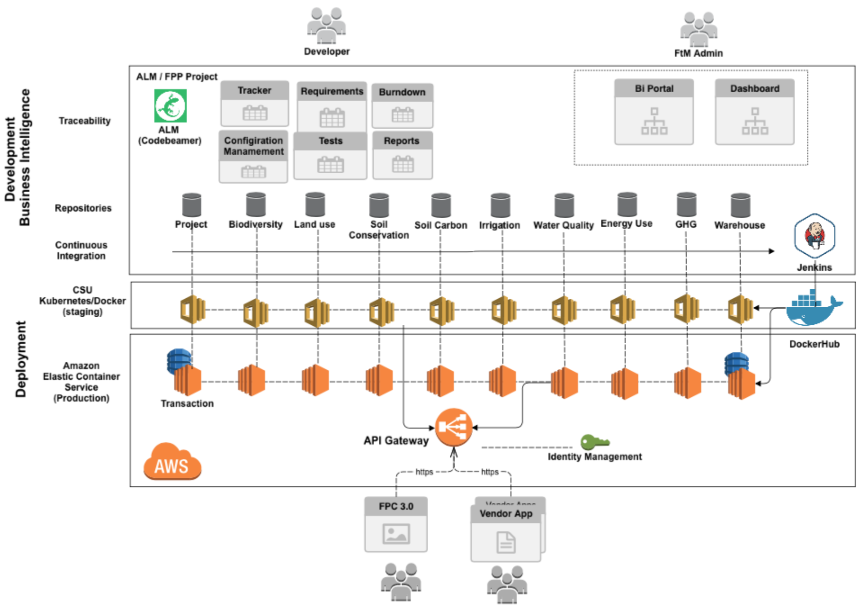


Figure 19: CSIParchitecture, credit David et al. (2014a).

CSIP elevates the MaaS concept by *"(1) accounting for modeling service elasticity and scalability, (2) leveraging contemporary computational approaches, (3) addressing traceability within operational settings, and (4) allowing platform and language agnostic service access and cloud agnostic deployments while providing a simple, non-invasive approach to leverage legacy and new modeling solution with minimal development effort"* (David et al. (2014a)) (Figure 19).

CSIP architectural design doesn't rely on a web-service central management, which facilitates provision of failover and redundancy features. Consequently, a large number of users can potentially perform several parallel model execution each without performance degradation.

CSIP interfaces with its MaaS services through a simple and open API, which adheres to RESTful API architectural design concepts (Fielding and Taylor (2002)). The service-client communication of model input and output parameters happens via JSON data objects.

The CSIP project was established and is actively developed at Colorado State University (CSU) in partnership with USDA-NRCS and USDA-ARS. CSIP currently hosts 250+ active environmental models and data services. CSIP-services can be easily accessed, tested and finally integrated into service delivery organizations information system workflows since they are deployed to OMSLab staging backends.

Catalog of inventoried services is currently growing and made available at `https://alm.engr.colostate.edu/cb/project/csip`.

For the sake of example, some of the available model services are: WEPP, WEPS, RUSLE2 (water and wind erosion prediction); SCI, STIR (soil quality); SWAT (SWAT-CP & SWAT-DEG), AgES-W, WQM, CFA (water quality and stream degradation); PRMS, NRCS Hydrotools (hydrology, water supply forecasting, stream hydraulics and sediment transport); GRAS (grazing management).

Some of the available data services are: NRCS Soil Data Mart (soil data); CLIGEN, WINDGEN, PRISM (climate data); ESIS/EDIT (ecological site data); WQM (pesticides data); LMOD (land management and crop rotations data).

Relevant representative cases of CSIP-services integration into service delivery organization workflows are: NRCS Integration Erosion and Resource Stewardship applications, which is the next generation Conservation Desktop application designed over CSIP-services access through NRCS API gateway; Wastewater Treatment Plant (WWTP), in which Colorado Department of Public Health and Environment (CDPHE) runs Flow Analysis (CFA) against USGS LOADEST and flood regression analysis models for monitoring regulations; National Cooperative Highway Research Program (NCHRP), which studies sediment transport and yield in water bodies (David et al. (2014a)).

3.3.2.3.1 CSIP API

CSIP interfaces with its MaaS services through a simple and open API, which adheres to RESTful API architectural design concepts (Fielding and Taylor (2000, 2002); David et al. (2014a)). REST architecture is deeply described in Section 3.3.2.2 *Microservice architecture and RESTful API*.

The service-client communication of model input and output parameters happens via JSON data objects.

CSIP REST protocol regulates access to CSIP-services. Although this protocol is similar to OpenGIS Consortium Web Processing Services (WPS) (Consortium et al. (2007)), data definitions, data descriptions, and meta-data are simplified to facilitate web-service access (David et al. (2014a)). A client is allowed to perform three operations against CSIP API through CSIP implementation:

1. Request a list of operative model and data services, which returns back service meta-data and implementation specifics;

2. Request model or data service details on a specific operation (e.g. input requirement and format);

3. Execute the model or data service by satisfying input parameter requirements and specifying optional metadata to control service execution.

Execution request can be submitted as synchronous (*"sync"*) or asynchronous (*"async"*). Afterwards, the model service state change to *"Running"*, and can (i) complete successfully and return *"Finished"*, (ii) fail and return "Failed", or (iii) get cancelled by the client and return *"Cancelled"* (David et al. (2014a)).

Listing 3.8 shows a REST call to EFH2 service: service endpoint (*POST*); target host (*Host*), metadata for proper JSON content negotiation (*Accept and Content-Type*). Listing 3.9 shows the service response, which includes the original request (field *"parameter"*) and the *"result"* field conforming REST standard. The *"metainfo"* field lists model run metadata.

Listing 3.8: REST call to EFH2, credit David et al. (2014a).

```
POST /csip-eft/m/efh2/1.1 HTTP/1.1
Host: <host>
Accept: application/json
Content-Type: application/json
```

Input files can be attached to a HTTP POST CSIP request as multipart form data attachments. The *"result"* section of a service run response may contain a URL pointing to additional model outputs. HTTP GET CSIP request against that URL allows for retrieving the additional model output in their legacy output format. HTTP GET CSIP request against the service endpoint returns a request *"template"*.

Listing 3.9: JSON response from a CSIP-eft run, credit David et al. (2014a).

```
{
  "metainfo": {
    "status": "Finished",
    "first_poll": 1000,
    "next_poll": 1000,
    "suid": "c6808036-db0b-11e3-84c6-8d184928a57a",
    "tstamp": "2014-02-14T02:02:17+0000",
    "service_url": "http:\/\/localhost:8080\/csip-eft\/m\/efh2\/1.0",
    "cpu_time": 16,
    "expiration_date": "2014-05-14T02:07:17+0000"
  },
  "parameter": [
    {
      "name": "precip",
      "description": "precip",
      "unit": "inch",
      "min": "1",
      "max": "15",
      "value": 14
    },
    {
      "name": "runoffcurvenumber",
      "min": "40",
      "max": "95","value": 90
    },
    {
      "name": "stormtype",
      "value": "I"
    },
    {
      "name": "watershedlength",
      "unit": "ft",
      "min": "200",
      "max": "26000",
      "value": 1500
    },
    {
      "name": "watershedslope",
      "unit": "%",
      "min": "0.1",
      "max": "64",
      "value": 0.5
    }
  ],
  "result": [
    {
      "name": "runoff",
      "value": 12.75
    },
    {
      "name": "timeofconcentration",
      "value": 0.727178
    },
    {
      "name": "unitpeakdischarge",
      "value": 0.3700218
    }
  ]
}
```

CSIP services are implemented against CSIP core library, which takes advantage of Java Jersey RESTful framework, and JAX-RS reference implementation (Burke (2013)). Listing 3.10 shows the service implementation of EFH2 model (USDA (1987)).

Listing 3.10: CSIP-compliant EFH2 model, credit David et al. (2014a).

```java
@Name( "EFH2")
@Description( "Storm runoff model based on " +
              "conventions in Engineering Field Handbook.")
@Path( "m/efh2/1.0")
@Polling(first = 1000, next = 1000)
public classV1_0 extends AbstractModelService {
  // the model
  EFH2HydrologyModel model = newEFH2HydrologyModel();

  @Override
  protectedString process()throwsException {
    Map m = getParamMap();
    model.setPrecip(m.get("precip").getDouble(VALUE));
    model.setRunoffCurveNumber(
          m.get("runoffcurvenumber").getInt(VALUE));
    // ... obtain more parameter here
    return model.simulate() == 0 ? EXEC_OK : EXEC_FAILED;
  }

  @Override
  protected JSONArray createResults() throwsException {
    JSONArray result = newJSONArray();
    result.put(JSONUtils.data("runoff",
                              model.getRunoffQ()));
    result.put(JSONUtils.data("timeofconcentration",
                              model.getTimeOfConcentration()));
    result.put(JSONUtils.data("unitpeakdischarge",
                              model.getUnitPeakDischarge()));
    return result;
  }
}
```

3.3.2.4 *Encog*

Encog is an open source machine learning framework developed by Jeff Heaton and released under Apache License version 2.0. The Encog library is available for Java and C♯, and it has been designed to pursue high scalability and adaptability to exploit multicore processors (Heaton (2015)).

This framework provides a large variety of machine learning models that can be used for regression, classification, and clustering purposes. The software is accurately designed and the exposed API allows for easily and quickly interchange machine learning methodology used with few or no code modifications.

The multithreaded implementation of the main training algorithms is finely tuned and empirical benchmarks show how Encog outperforms many concurrent Java and C♯ libraries (Taheri (2010); Ramos-Pollán et al. (2012); Iuhasz et al. (2013)).

The library is well documented and about 150 examples are provided to exercise most of the API capabilities (Heaton (2015)).
The available machine learning models are following listed:

- Adaline, Feedforward, Hopfield, PNN/GRNN, RBF, FS-NEAT and Hyper-NEAT neural networks;
- Generalized linear regression (GLM);
- Genetic programming;
- K-means clustering;
- K-nearest neighbors;
- Linear regression;
- Self-organizing map (SOM);
- Simple recurrent network (Elman and Jordan);
- Support vector machine (SVM).

Optimization algorithms to minimize a loss function (e.g. particle swarm optimization, Nelder-Mead and simulated annealing) along with preprocessing tool for training, test, and validation data set splitting are provided.
Encog allows for storing and reloading a trained artificial neural network leveraging Java serialization.

3.3.2.5 *Surrogate Model Services implementation*

FeNS is implemented as a set of 5 CSIP microservices: *collect*, *normalize*, *train*, *select*, and *run*. User calls the services in the previous order to (1) collect and store model input/output snapshots into an application dedicated NoSQL database, (2) normalize the uploaded dataset, (3) train, validate, and store a collection of ANNs, (4) select the utmost performant ones to build an ensemble of ANNs, and (5) run the ensemble of ANNs against new data. Figure 20 illustrates the FeNS CSIP-services pipeline workflow: the left side (*SM GENERATION*) is actually hidden to the end user who interacts with the CSIP-run service only (right hand side, *SM APPLICATION*).
CSIP microservices have been designed on the following principles:

- simplify user-service interface by exposing strictly necessary parameters;
- avoid non essential data transfer from database to service and vice-versa;
- interact with database collections to track data and SMs metadata over the FeNS services pipeline, and enrich them with additional information over service run.

Users interface with each web-service by posting a JSON payload. Each payload is composed of two sections to comply with CSIP API: *metainfo*, and *parameter*. The metainfo section lists information to control a service run (e.g. ``mode'': ``async'' to request asynchronous run). The parameter section contains an array of JSON Objects. Each JSON Object contains two <key>:<value> tuple with predefined keys: <name>: <service_var_name> and <value>: <value_to_assign>. Some fields of the CSIP-collect JSON payload might have an additional

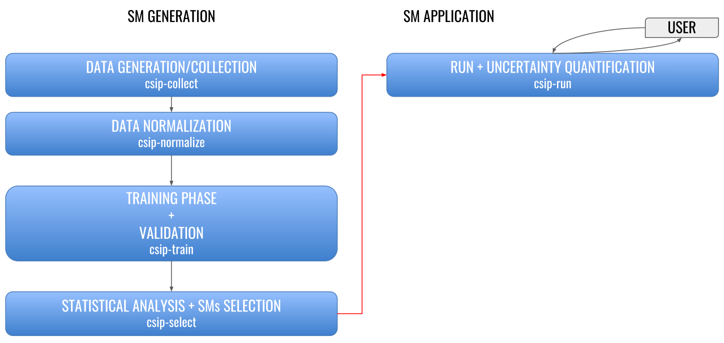


Figure 20: Set of 5 CSIP services that allows for generating the SM.

`<description>: <var_description>` tuple. Section 3.3.2.5.1 *Service 1: raw data collection* explains purpose and usage of this extra field.

Each JSON payload set up service-specific input parameters. Consequently, every JSON payload contains a different set of JSON Objects. User can retrieve a payload template of each service by using a GET request.

This subsection is organized as follow: Section *Service 1: raw data collection* describes the collection of raw data into dedicated database per snapshot or by attaching a csv file of snapshots to the JSON request; Section *Service 2: data normalization* introduces to the computationally cost-effective implementation of the normalization aggregator pipeline; Section *Service 3: SM creation*analyzes the core service walking through TV dataset splitting, SM training, SM validation, and SM store phases; Section *Service 4: SMs selection, building the ensemble* describes implemented strategies to select the most performant SMs and build the eSM; Section *Service 5: eSM run* shows post and response of an ensemble run.

Each subsection provides service sequence diagram, service payload template, and database formal structure design. Service implementation code snippets and Unified Modeling Language (UML) diagram are presented when necessary to integrate service algorithm description.

3.3.2.5.1 Service 1: raw data collection

The first service collects the raw TV dataset into a dedicated MongoDB collection for later ANN training. It manages the upload of a single snapshot of input/output model data as well as a csv file containing a list of snapshots.

Figure 21 shows the interactive behaviour of the CSIP-collect service through a sequence diagram.

A data-driven SM learns and traps the knowledge of a conceptual/physical model by capturing the nonlinear model behaviour hidden into an input/output snapshot.

Thus, the collection of big number of combinations of model input/output is the first step towards the creation of an SM. CSIP-collect service stores data and related metadata into a `raw` MongoDB database collection.

CSIP-collect service provides two options to parse and store raw data and related metadata:

1. **per snapshot:** collection of single input/output snapshot;

2. **per csv file:** collection of numerous input/output snapshots from a csv file.

The two options are following separately analyzed.

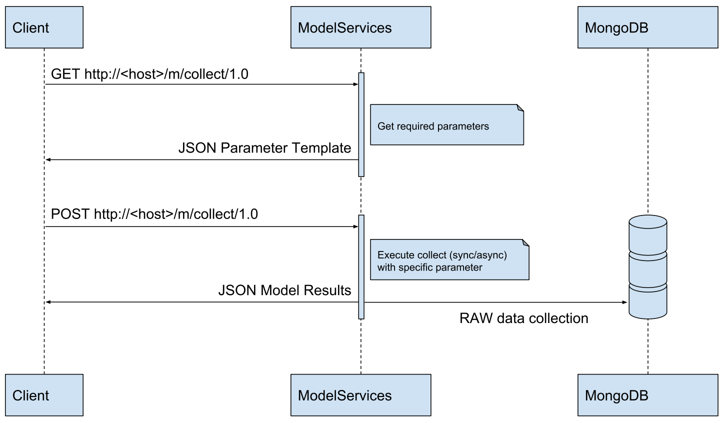


Figure 21: Sequence diagram of CSIP-collect service.

Collection per snapshot

Collecting data per snapshot requires a single combination of original model input/output. Listing 3.11 shows a CSIP-collect JSON request template.

Listing 3.11: Template JSON payload of CSIP-collect.

```
{
  "metainfo": {},
  "parameter": [
    {
      "name": "annName",
      "value": "db_name"
    },
    {
      "name": "in_var1",
      "value": 0.1,
      "description": "in"
    },
    {
      "name": "in_var2",
      "value": 19,
      "description": "in, [0.1,0.9]"
    },
    {
      "name": "in_var3",
      "value": 0,
      "description": "in, normalized"
    },
    {
      "name": "out_var1",
      "value": 0.28,
      "description": "out"
    },
    {
      "name": "out_var2",
      "value": 141.893,
      "description": "out, normalized"
    },
    {
      "name": "out_var3",
      "value": 0.081694489550937,
      "description": "out, [-1, 1]"
    }
  ]
}
```

The model parameter section is an array that contains a list of JSON objects.

The first object contains two `<key>: <value>` tuple: `<name>: <annName>` and `<value>: <db_name>`. Here, the CSIP-collect service checks if the database `db_name` already exists: if it does exist, the snapshot of data is pushed to the bottom of the `raw` collection; if it doesn't exist, an new database and a `raw` collection are created.

The following JSON objects are the list of variables required for training the surrogate model. Every variable object contains the model variable name, its value and description. The latter is of key importance since it contains comma separated list of variable metadata required by following services in the FeNS pipeline. The description field can contains three type of information 3.11.

1. Type (*in*/*out*): it describes if the variable is input or output of the supervised learning process;
2. Normalization (*normalized*): it tells if the variable is normalized already;
3. Normalization min and max ($[norm_min, norm_max]$): the range the variable has to be normalized in.

Only the Type metadata is mandatory. If it is not provided, CSIP-collect skips that variable and doesn't push it to the database. If the other two fields are missing, default values are assumed: *normalized=false* (variable requires normalization), *norm_min=0* and *norm_max=1*.

Collection per csv file

If a csv file is attached to the JSON payload, CSIP-collect service enables a different input/output snapshots parsing. The structure of a basic JSON payload remains identical to Listing 3.11 . Here, CSIP-collect service reads only variable names and description, and skips value fields. Afterwords, CSIP-collect processes the csv file header and uploads to the database only columns which names are listed in the JSON payload. CSIP-collect implements a collection algorithm which allows for pushing to the database chunks of 10000 snapshots at a time. This very effective design allows for collecting hundred of thousands of snapshots in few seconds.

By default data provided to CSIP-collect service (per snapshot or per csv file) are considered the previously called TV dataset (Section *Ensemble of surrogate models and uncertainty quantification*). CSIP-train service uses the normalized TV dataset to randomly creates training and validation datasets. Consequently, each SM is trained on a different dataset but also validate on a different dataset. When it comes to selecting the most performant SMs, CSIP-select goes through validation statistics of each trained SM and select the most accurate. However, each SM is validated on a different dataset, thus CSIP-select comparison are not *"perfectly"* fair.

The described methodology is acceptable when user deals with small dataset. If the number of snapshots gathered in the csv file is of considerable size, user may want to split them into two subsets and store them into separated collections: a TV dataset for SMs training; and a further validation dataset which is used by the CSIP-train service to commonly validate every trained SM against. This allows CSIP-select to actually compare SM validation performances computed on the identical dataset.

Detailed explanation of how datasets are split and used by CSIP-train service is available at Paragraph *Service 3: SM creation*. Listing 3.12 highlights the two extra JSON objects required to enable this additional dataset splitting.

Listing 3.12: Template JSON payload of CSIP-collect to generate a common dedicated *"validation"* collection.

```json
{
  "metainfo": {},
  "parameter": [
    {
      "name": "annName",
      "value": "name of the surrogate model"
    },
    {
      "name": "split",
      "value": "true"
    },
    {
      "name": "training_perc",
      "value": "0.8"
    },
    {
      "name": "variable 1",
      "value": 0.1,
      "description": "in"
    },
    {
      "name": "variable 2",
      "value": 0 ,
      "description": "in, normalized"
    },
    {
      "name": "variable 3",
      "value": 141.893,
      "description": "in, [0.1,0.9]"
    },
    {
      "name": "variable 4",
      "value": 0.081694489550937,
      "description": "out, [-1, 1]"
    }
  ]
}
```

When *split* boolean variable is *true*, the splitting mechanism is enabled. It splits the uploaded data into TV and common validation dataset based on a user-defined percentage, making sure that the two datasets have similar probability distribution. It also checks that min and max values for each variable are part of the TV dataset.

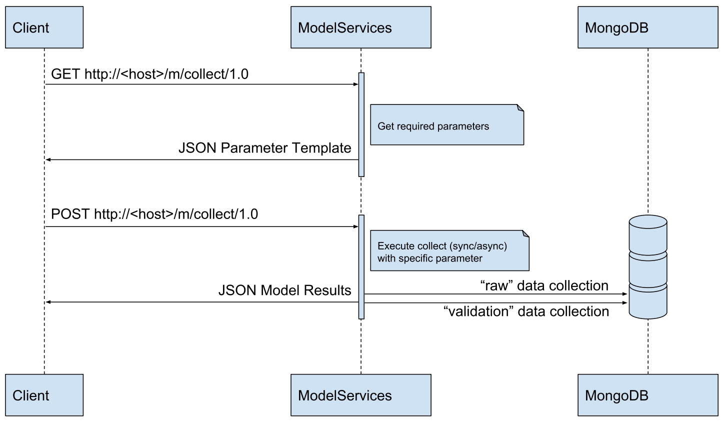
TV dataset is always stored into the so called *"raw"* collection either using per snapshot or per csv file collection. If split option is enabled, the common validation dataset is pushed into the dedicated *"validation"* collection.

Figure 22 shows the interactive behaviour of the CSIP-collect service through a sequence diagram when split option is enabled. Here, TV dataset and common validation dataset are stored into *"raw"* collection and *"validation"* collection respectively.

MongoDB: formal structure

CSIP-collect service stores each original model variable as a BSON Document. Figure 23 shows the formal structure of the *"raw"* collection; if *"validation"* collection is created, it has identical formal structure.

MongoDB automatically creates *id_* (ObjectId is the unique immutable primary key that identifies each document) and *timestamp* (Date object holds the time the collection is created). *values* is the actual array of collected raw data. *metadata* contains a list of fields that fully describe the model variable: variable *name*; in/out



Client
ModelServices
MongoDB
GET http://<host>/m/collect/1.0
Get required parameters
JSON Parameter Template
POST http://<host>/m/collect/1.0
Execute collect (sync/async) with specific parameter
JSON Model Results
"raw" data collection
"validation" data collection
Client
ModelServices
MongoDB

Figure 22: Sequence diagram of CSIP-collect service when csv file is attached.

if the variable is model input or output (*type*); true/false if the raw data is normalized already (*norm*); the lowest boundary for normalization algorithm (*norm_min*); the highest boundary for normalization algorithm (*norm_max*); the actual *min* and *max* values in the provided raw data array; and a *values_id* which is identical to every variable and changes only when CSIP-collect service pushes additional data to the collection. CSIP-select services uses *values_id* field to invalidate old SMs. Detailed description is provided in Paragraph 3.3.2.5.4 *Service 4: SMs selection, building the ensemble*.

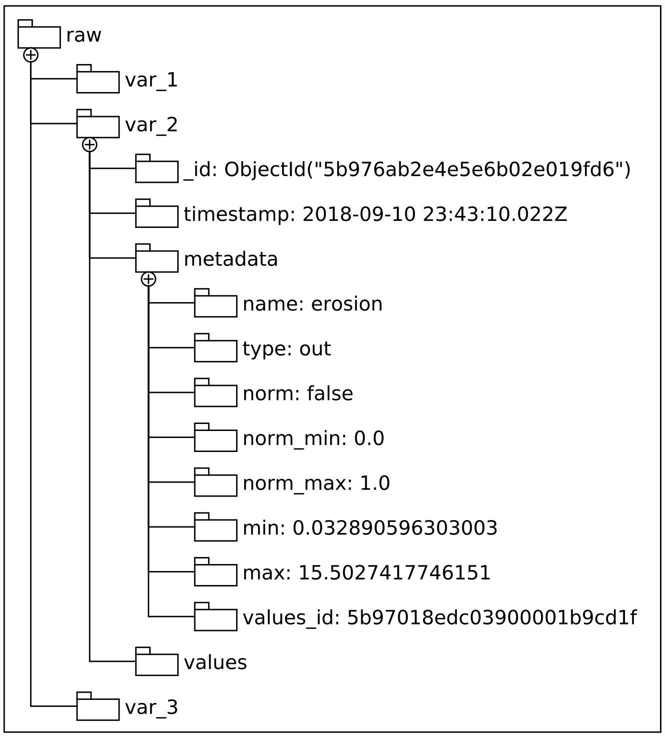


Figure 23: Formal structure of MongoDB *"raw"* collection.

3.3.2.5.2 Service 2: data normalization

The second service normalizes the TV dataset collected into *"raw"* MongoDB collection and stores normalized results into a dedicated collection called *"normalized"*.

Figure 24 shows the interactive behaviour of the CSIP-normalize service through a sequence diagram.

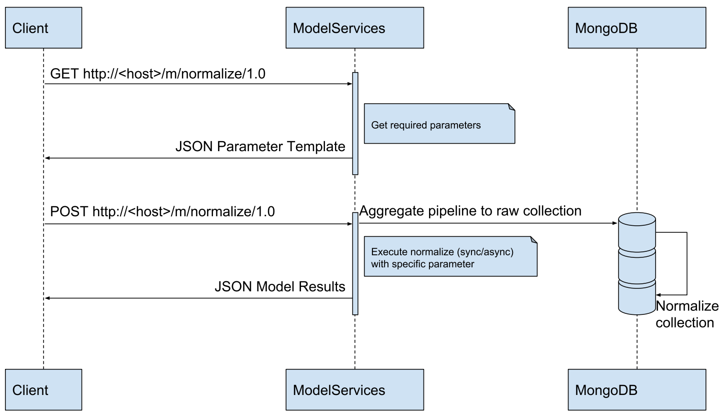

Client
ModelServices
MongoDB
GET http://<host>/m/normalize/1.0
Get required parameters
JSON Parameter Template
POST http://<host>/m/normalize/1.0
Aggregate pipeline to raw collection
Execute normalize (sync/async) with specific parameter
JSON Model Results
Normalize collection
Client
ModelServices
MongoDB

Figure 24: Sequence diagram of CSIP-normalize service.

The CSIP-normalize service works as gateway to the database. Since CSIP-normalize doesn't perform any operation on the data and Feature Scaling is the only normalization algorithm currently available, the JSON payload requests is really minimal.

Listing 3.13: Template JSON payload of CSIP-normalize service.

```
{
  "metainfo": {},
  "parameter": [
    {
      "name": "annName",
      "value": "db_name"
    }
  ]
}
```

CSIP-normalize service integrates MongoDB aggregation operations to perform arithmetic expressions on grouped data records database-side. This allows for avoiding raw data copy from database to service and normalized data vice versa. As a result, CSIP-normalize service sole responsibility is to build the aggregation operator and push it to the database.

MongoDB provides three different aggregation operators: single purpose aggregation methods, map-reduce function, and aggregation pipeline.

Single purpose aggregation operations and map-reduce function are not part of the CSIP-normalize normalization pipeline but are briefly introduced for the sake of completeness.

Single purpose aggregation operations simply performs aggregation operations on documents of an entire collection. Its use is pretty straightforward but lacks of flexibility and provides pretty limited functionalities such as count and distinct operations applied to documents that matches a `find()` query (MongoDB (2019a)).

Map-reduce function performs a *mapping* of each selected document of the collection (i.e. documents that matches a `find()` query) and emits document key-value pairs (MongoDB (2019b)). If a key has multiple values, the key is *reduced*:

MongoDB collects and condenses the aggregated data and returns a document or writes results to a collection (MongoDB (2019b)). JavaScript is utilized to perform custom map-reduce functions. This allows for higher flexibility with respect to single purpose aggregation and aggregation pipeline. Contemporary, it increases complexity and ineffectiveness.

CSIP-normalize normalization pipeline is an aggregation pipeline. The latter is a framework designed upon the idea of processing documents off of a single collection in a multi-stage pipeline (MongoDB (2019b)). The entire collection passes through the pipeline, and every stage performs one in-memory operation. A stage may filter out documents, generate new ones or transform them. Especially the last operation is of particular interest for the normalization pipeline. Transformations happen because of pipeline expressions which might be arithmetic expressions, array expressions, text expressions, etc (MongoDB (2019b)). Expressions operate on fields of input documents. Because of their flexible syntax, nesting expressions is allowed.

Before describing the Java MongoDB client syntax that implements the normalization pipile, the Feature Scaling equation is presented:

$$f(x) = \frac{(x - d_L) * (n_H - n_L)}{d_H - d_L} + n_L, \tag{3.3}$$

where x is the value to normalize, d_H is the maximum value in the array, d_L is the minimum value in the array, n_H and n_L the maximum and minimum values to normalize data in.

Listing 3.14 shows the Java MongoDB client syntax that implements the Feature Scaling equation. The normalization pipeline concurrently operates on each document of the collection.

Listing 3.14: MongoDB Java client of Feature Scaling aggregator pipeline.

```java
asList(
  // stage 1
  project(fields(
    excludeId(),
    include("values", "metadata.name", "metadata.type",
      "metadata.norm", "metadata.norm_min", "metadata.norm_max",
      "metadata.min", "metadata.max", "metadata.values_id"),
    computed("count", new Document("$size", "$values")),
    computed("range", new Document(
      "$subtract", asList("$metadata.max", "$metadata.min")))
  )),
  // stage 2
  unwind("$values"),
  // stage 3
  project(fields(
    excludeId(),
    include("values", "min", "max", "count",
      "metadata.name", "metadata.type", "metadata.norm",
      "metadata.norm_min", "metadata.norm_max", "metadata.values_id"),
    computed("normVals", new Document(
      "$cond", asList("$metadata.norm", "$values", new Document(
        "$sum", asList(new Document(
          "$divide", asList(new Document(
            "$multiply", asList(new Document(
              "$subtract", asList("$values", "$min")),new Document(
                "$subtract", asList("$metadata.norm_max",
                                    "$metadata.norm_min"))
            )
          ), "$range")
        ), "$metadata.norm_min")
      ))
    ))
  )),
  // stage 4
  group(new Document("name", "$metadata.name")
    .append("type", "$metadata.type")
    .append("norm", "$metadata.norm")
    .append("norm_min", "$metadata.norm_min")
    .append("norm_max", "$metadata.norm_max")
    .append("values_id", "$metadata.values_id")
    .append("count", "$count")
    .append("min", "$min")
    .append("max", "$max"), push("values", "$normVals")),
  // stage 5
  project(fields(
    excludeId(),
    computed("timestamp", new Date()),
    computed("metadata", new Document("name", "$_id.name")
      .append("type", "$_id.type")
      .append("norm", "$_id.norm")
      .append("norm_min", "$_id.norm_min")
      .append("norm_max", "$_id.norm_max")
      .append("values_id", "$_id.values_id")
      .append("count", "$_id.count")
      .append("min", "$_id.min")
      .append("max", "$_id.max")
      .append("min_index", new Document(
        "$indexOfArray", asList("$values", "$_id.norm_min")))
      .append("max_index", new Document(
        "$indexOfArray", asList("$values", "$_id.norm_max")))),
    computed("values", "$values"))
  ),
  // stage 6
  out(to_collection)
);
```

Listing 3.14 already highlights that the entire pipeline is split into 6 stages: Stage 1 implements a project operator as well as Stage 3 and Stage 5; Stage 2 implements an unwind operator; Stage 4 implements a group operator; Stage 6 implements an out operator. Figure 25 shows the schematic of the normalization pipeline. Each rectangle represents a BSON document. Stage 1 operates on a single BSON document. Stage 2 explodes the input single BSON document to operate on its nested BSON documents. Stage 3 gets a list of BSON documents from the previous Stage and combines them. Stage 4 and 5 operates on a single BSON document, while Stage 6 takes care of transferring Stage 5 output to a new MongoDB collection.

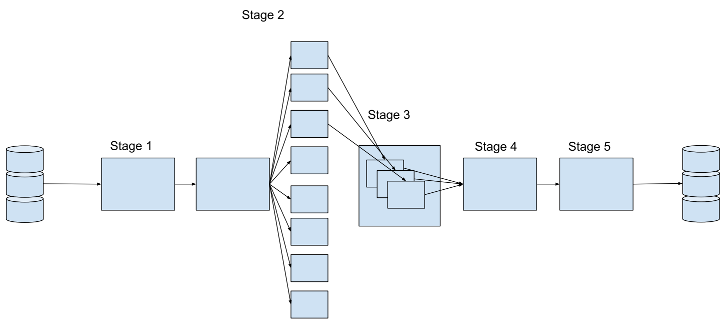


Figure 25: Schematic of the Stages involved in the Feature Scaling aggregator pipeline.

Stage 1 is a projection operator (Listing 3.15): it analyzes values and metadata of the input document (on original model variable) and returns a single BSON document with fields required to compute feature scaling expression exclusively. This limits the scope of the data query improving computational speed and efficiency.

Listing 3.15: Stage 1 code snippet.

```
3   project(fields(
4     excludeId(),
5     include("values", "metadata.name", "metadata.type",
6       "metadata.norm", "metadata.norm_min", "metadata.norm_max",
7       "metadata.min", "metadata.max", "metadata.values_id"),
8     computed("count", new Document("$size", "$values")),
9     computed("range", new Document(
10      "$subtract", asList("$metadata.max", "$metadata.min")))
11  )),
```

The *_id* of the input document is excluded by `excludeId()` method, which is included in the output BSON by default otherwise.

The `include()` method groups fields already available from the input document into one projection (the array of *values*, *metadata.name*, *metadata.type*, *metadata.norm*, *metadata.norm_min*, *metadata.norm_max*, *metadata.min*, *metadata.max*, *metadata.values_id*). These fields may be necessary to perform the normalization operation or just important metadata to carry over to the normalized collection.

Two additional fields are actually created during the project Stage and included into the BSON document: *count* and *range* (Listing 3.15 - Line 8,9). The latter are created using the `computed()` method, which returns the computed value of the given expression.

The *count* field is the number of elements in the *values* array. *size* is the MongoDB predefined operator that carries out this task.

The *range* field is the *subtraction* of max and min values in the raw data array, and is following used as denominator of the Feature Scaling expression.

Stage 2 performs the `unwind()` operation (Listing 3.16).

Listing 3.16: Stage 2 code snippet.

```
unwind("$values"),
```

In order to enable actual computation on each value of the values array, the BSON document output from Stage 1 is disassembled: one document per value is extracted and created in memory.

Stage 3 operates the actual normalization algorithm on each document output of the `unwind` stage (Listing 3.17).

Listing 3.17: Stage 3 code snippet.

```
project(fields(
  excludeId(),
  include("values", "min", "max", "count",
    "metadata.name", "metadata.type", "metadata.norm",
    "metadata.norm_min", "metadata.norm_max", "metadata.values_id"),
  computed("normVals", new Document(
    "$cond", asList("$metadata.norm", "$values", new Document(
      "$sum", asList(new Document(
        "$divide", asList(new Document(
          "$multiply", asList(new Document(
            "$subtract", asList("$values", "$min")),new Document(
              "$subtract", asList("$metadata.norm_max",
                                  "$metadata.norm_min"))
          )
        ), "$range")
      ), "$metadata.norm_min")
    ))
  ))
)),
```

For each processed document, the *_id* field is excluded (`excludeId()` method) while metadata (*metadata.name*, *metadata.type*, *metadata.norm*, *metadata.norm_min*, *metadata.norm_max*, *metadata.values_id*) and relevant fields (*values*, *min*, *max*, *count*) are included in the document that is sent over to the following stage.

Stage 3 actually computes the normalized value by applying the normalization equation to the input value (Listing 3.17). *normVals* is following added to the output BSON document.

The core of the algorithm from the most nested to the most external module (Listing 3.17): (1) subtracts *values* and *min*, and *metadata.norm_max* and *metadata.norm_min*; (2) multiplies the two subtractions; (3) divides the multiplication by *range*; and (4) finally sums the division to *metadata.norm_min*.

The most external module is the cond conditional expression operator. It is a ternary operator that checks *metadata.norm* field to avoid the normalization of an already normalized input dataset (Listing 3.18). *metadata.norm* is a true boolean field if values of the variable stored in the document involved is already normalized, false otherwise.

Listing 3.18: Conditional operator leveraged in Stage 3.

```
new Document("$cond", asList("$metadata.norm",
                              "$values", <NORMALIZATION>))
```

Every ouput BSON document from Stage 3 contains a single normalized value.

Stage 4 makes use of `group()` operator to merge documents with a single normalized value into one document gathering the array of normalized values (Listing 3.19).

Listing 3.19: Stage 4 code snippet.

```
group(new Document("name", "$metadata.name")
  .append("type", "$metadata.type")
  .append("norm", "$metadata.norm")
  .append("norm_min", "$metadata.norm_min")
  .append("norm_max", "$metadata.norm_max")
  .append("values_id", "$metadata.values_id")
  .append("count", "$count")
  .append("min", "$min")
  .append("max", "$max"), push("values", "$normVals")),
```

Here, the push operator plays a key role: it pulls normalized value *normVals* from each document output of Stage 3 and pushes them into the array *values* of the new single BSON document. The latter is the only output of Stage 4.

Stage 5 simply adds few extra metadata to the incoming BSON document: *timestamp* generated using the Date object, index of min and max values in the *values* array through the *indexOfArray* expression (Listing 3.20).

Listing 3.20: Stage 5 code snippet.

```
project(fields(
  excludeId(),
  computed("timestamp", new Date()),
  computed("metadata", new Document("name", "$_id.name")
    .append("type", "$_id.type")
    .append("norm", "$_id.norm")
    .append("norm_min", "$_id.norm_min")
    .append("norm_max", "$_id.norm_max")
    .append("values_id", "$_id.values_id")
    .append("count", "$_id.count")
    .append("min", "$_id.min")
    .append("max", "$_id.max")
    .append("min_index", new Document(
      "$indexOfArray", asList("$values", "$_id.norm_min")))
    .append("max_index", new Document(
      "$indexOfArray", asList("$values", "$_id.norm_max")))),
  computed("values", "$values"))
),
```

Stage 6 writes the input documents into *"normalized"* MongoDB collection (Listing 3.21).

Listing 3.21: Stage 6 code snippet.

```
out(to_collection)
```

The MongoDB aggregation framework provides very limited mathematical operators. However, javascript functions can be stored on the database server and used through the map-reduce aggregation operators.

"validation" collection normalization

CSIP-normalize service checks if *"validation"* collection exists in the database. If the collection is available, CSIP-normalize pushes the just described normalization pipeline to this collection as well. Normalized results are stored into *"validNorm"* collection. Figure 26 shows the interactive behaviour of the CSIP-normalize service through a sequence diagram when both *"raw"* and *"validation"* collection are available. Compared to Figure 26, here two aggregation pipelines are sent to MongoDB database, and two normalized collection are created consequently.

Even if both collection are available and two normalization processes are performed, the aggregation pipeline is very effective and requires few seconds for normalizing hundred of thousands of snapshots.

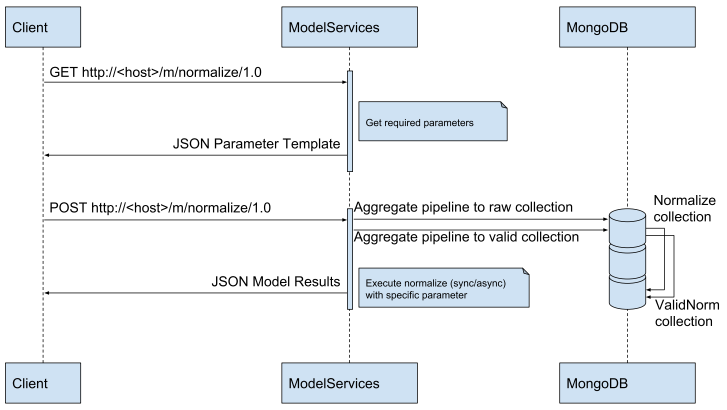

Client
ModelServices
MongoDB
GET http://<host>/m/normalize/1.0
Get required parameters
JSON Parameter Template
POST http://<host>/m/normalize/1.0
Aggregate pipeline to raw collection
Aggregate pipeline to valid collection
Normalize collection
JSON Model Results
Execute normalize (sync/async) with specific parameter
ValidNorm collection
Client
ModelServices
MongoDB

Figure 26: Sequence diagram of CSIP-normalize service when *"raw"* and *"valid"* collection are available.

MongoDB: formal structure

CSIP-normalize service stores each normalized variable as a BSON Document. Figure 27 shows the formal structure of the *"normalized"* collection; if *"validation"* collection is available, a *"validNorm"* collection is created with identical formal structure.

With respect to *"raw"* collection formal structure (Figure 23), *"normalized"* metadata lists three additional fields: *count* of the number of elements in the array, and index of min and max values in the array (*min_index*, *max_index*).

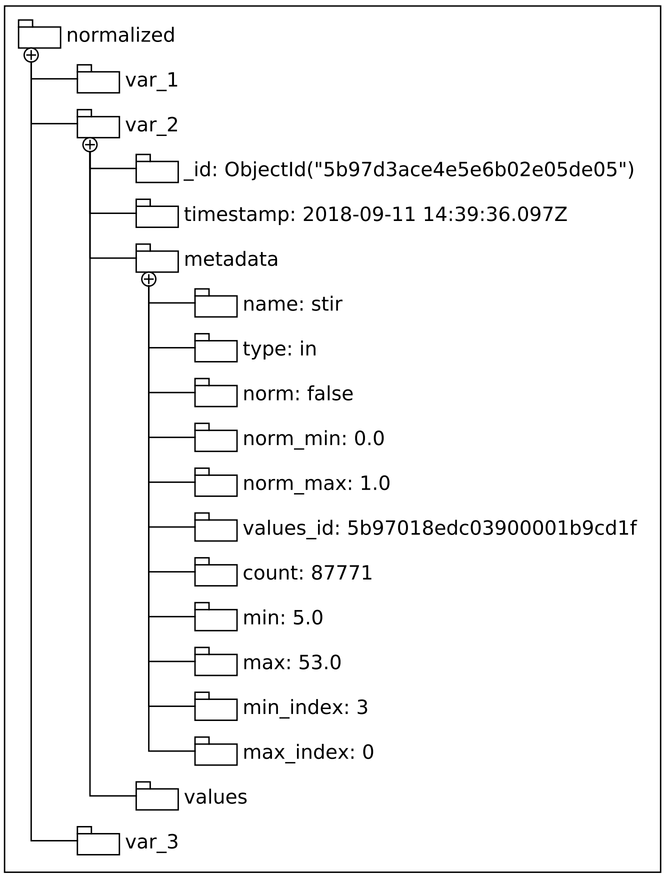


Figure 27: Formal structure of MongoDB *"normalized"* collection.

3.3.2.5.3 Service 3: SM creation

The third service is the core of the FeNS pipeline since it creates the SM through supervised learning, evaluates SM goodness of fit and stores both structure and related metadata into the database.

A single run of CSIP-train service generates one SM. Consequently, CSIP-train service has to be invoked several times to create a collection of SMs (Figure 28).

For the sake of description, CSIP-train service workflow can be split into four steps:

1. TV dataset splitting;
2. SM training;
3. SM validation;

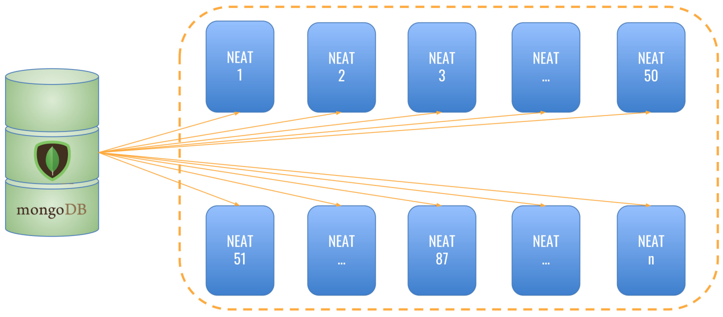


Figure 28: Conceptual approach of CSIP-train ensemble SMs.

4. SM store.

Figure 29 shows the interactive behaviour of the CSIP-train service through a sequence diagram. Here, the interaction ModelService - MongoDB database after a client POST request is crucial part of the service architectural design.

The interaction ModelService - MongoDB database is described through three arrows: top-arrow directed from MongoDB to ModelService represents the transferring of *"normalized"* collection from database to CSIP-train service; the second arrow directed from ModelService to MongoDB symbolizes the process of storing a validated but partially trained SM and its metadata; the third and last arrow directed from ModelService to MongoDB indicates the process of storing the validated SM and its metadata when the training phase is over.

The description of the ModelService - MongoDB interaction shows that SM training, validation, and store phases actually overlaps: during the training phase, user might decide to regularly validate and store a partially trained ANN to evaluate SM emulation performance, the so called *recovery* phase. Consequently, validation-store phases are potentially nested into SM training, and additionally run when the training phase is over.

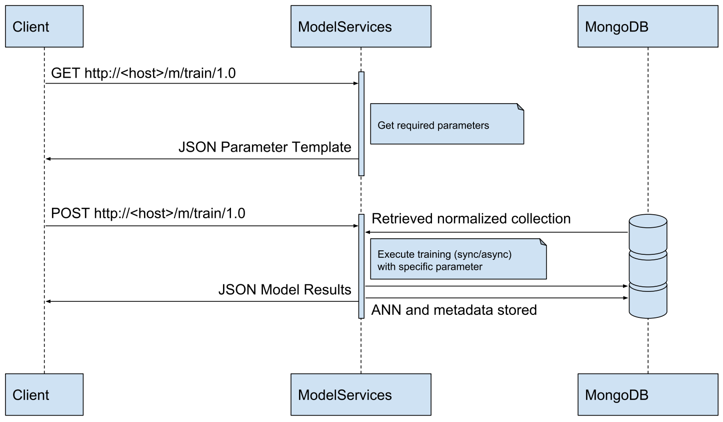

Client
ModelServices
MongoDB
GET http://<host>/m/train/1.0
Get required parameters
JSON Parameter Template
POST http://<host>/m/train/1.0
Retrieved normalized collection
Execute training (sync/async) with specific parameter
JSON Model Results
ANN and metadata stored
Client
ModelServices
MongoDB

Figure 29: Sequence diagram of CSIP-train service.

Along with application specific database name, the JSON request payload contains seven fields required to set up split mechanism, genetic algorithm, training thresholds, and recovery recurrence: *scale_mechanism* and *training_perc* allow for choosing the splitting algorithm and the percentage of training and validation datasets; *population* and *connection_density* set up initial number and structure of ANNs in the neuroevolutionary genetic algorithm; *training_error* and *max_epochs* define two training stopping criteria such as Mean Squared Error (MSE) threshold and maximum number of iterations (or epochs) respectively; *recovery_epochs* defines the recurrence of recovery phase (validation and store nested in the training phase) (Listing 3.22). Further details on splitting algorithms and training stopping criteria are provided in Paragraph *PHASE 1: TV dataset splitting* and *PHASE 2: SM training* respectively.

Every phase of CSIP-train service following analyzed and UML diagrams are provided when helpful to the description.

Listing 3.22: Template JSON payload of CSIP-train service.

```
{
  "metainfo": {},
  "parameter": [
    {
      "name": "annName",
      "value": "clay_test"
    },
    // splitting section
    {
      "name": "training_perc",
      "value": 0.8
    },
    {
      "name": "scale_mechanism",
      "value": "SameDistribution"
    },
    // NEAT set up
    {
      "name": "population",
      "value": 3000
    },
    {
      "name": "connection_density",
      "value": 1
    },
    // actual training set up
    {
      "name": "training_error",
      "value": 0.01
    },
    {
      "name": "max_epochs",
      "value": 5000
    },
    // storing set up
    {
      "name": "recovery_epochs",
      "value": 500
    }
  ]
}
```

PHASE 1: TV dataset splitting

The first step of the SM creation is the split of the normalized TV dataset from *"normalized"* collection. Here, the modeler may want to randomize the TV dataset before splitting it, or keep it in the provided order.

Consequently, CSIP-train service implements three split algorithms:

- **simple** algorithm splits TV dataset in training and validation on user-defined percentage;
- **random** algorithm randomizes TV dataset before splitting it in training and validation on user-defined percentage;

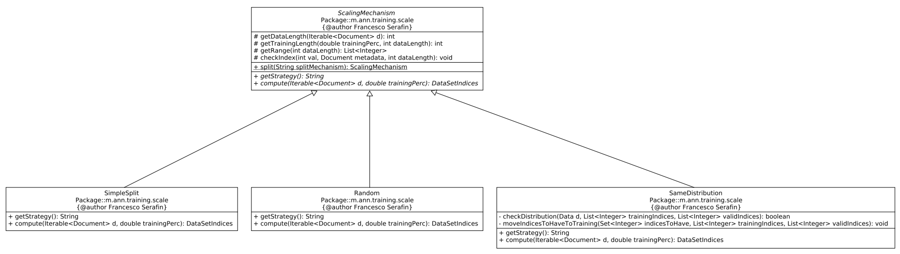

ScalingMechanism
Package::m.ann.training.scale
{@author Francesco Serafin}
# getDataLength(Iterable<Document> d): int
# getTrainingLength(double trainingPerc, int dataLength): int
# getRange(int dataLength): List<Integer>
# checkIndex(int val, Document metadata, int dataLength): void
+ split(String splitMechanism): ScalingMechanism
+ getStrategy(): String
+ compute(Iterable<Document> d, double trainingPerc): DataSetIndices
SimpleSplit
Package::m.ann.training.scale
{@author Francesco Serafin}
+ getStrategy(): String
+ compute(Iterable<Document> d, double trainingPerc): DataSetIndices
Random
Package::m.ann.training.scale
{@author Francesco Serafin}
+ getStrategy(): String
+ compute(Iterable<Document> d, double trainingPerc): DataSetIndices
SameDistribution
Package::m.ann.training.scale
{@author Francesco Serafin}
- checkDistribution(Data d, List<Integer> trainingIndices, List<Integer> validIndices): boolean
- moveIndicesToHaveToTraining(Set<Integer> indicesToHave, List<Integer> trainingIndices, List<Integer> validIndices): void
+ getStrategy(): String
+ compute(Iterable<Document> d, double trainingPerc): DataSetIndices

Figure 30: UML of implemented scaling mechanism.

- **same distribution** randomizes TV dataset, splits training and validation dataset making sure that snapshots containing min or max values of each variable are in the training dataset, compares probability distribution of training and validation dataset per variable and if are not similar restarts the algorithm.

User can set the *scale_mechanism* in the JSON payload to "simple", "random", or "samedistribution".

Splitting algorithm is designed by implementing Simple Factory principle through a static method (Freeman et al. (2004)). Additional splitting algorithms are easily implemented by extending *ScalingMechanism* abstract class and overriding the `getStrategy()` and `compute()` methods (Figure 30).

The `getStrategy()` method returns the name of the splitting mechanism, which becomes part of the SM metadata stored in the SM dedicated MongoDB collection. The compute method hosts the implementation of the actual algorithm and returns the *"random training"* and *"random validation"* datasets to the SM training method.

PHASE 2: SM training

The second step is the actual creation of the SM. Here, NEAT employes the *"random training"* dataset in a supervised learning procedure coupled to neuroevolutionary genetic algorithm to emerge the SM.

population field in the JSON payload defines the initial number of the ANNs that are concurrently generated by the neuroevolutionary genetic algorithm in one CSIP-train instance. Only the best genome survives the supervised learning process and becomes the SM.

NEAT supervised learning procedure is based on a stochastic approach. Consequently, the bigger the initial population the higher the chance to generate a more accurate SM. However, a wider population is computationally more expensive since the genomes concurrently mutates and evolves. CSIP-train service initial population default value is set to 1 000 units.

connection_density is the parameter that defines the initial connection frequency between input and output nodes of each genome in the FS-NEAT population.

Encog NEAT (Heaton (2015)) implements this algorithm with a nested for statement: the outer for statem loops over each input node while the inner for statement loops through every output node. Input node and output node get connected if a randomly generated number between 0 and 1 (excluded) is smaller than user-provided *connection_density* (Algorithm 1).

Algorithm 1: Pseudo-code of FS-NEAT population initialization in Encog.

```
Input: input nodes
Output: output nodes
foreach input node do
    foreach output node do
        if random value < connection_density then
            connect input to output nodes with random weight
        end
    end
end
```

With a connection density of 0.1, very few input nodes have initial probability of being connected to output nodes. With a connection density of 1.0, every input node is connected to each output node. The range of connection density values between 0.0 and 1.0 (excluded) activates FS-NEAT (Whiteson et al. (2005)) features: not every input node is connected to the output nodes and the neuroevolutionary genetic algorithm adds connection if and only if it improves genome accuracy.

FS-NEAT has proven to generate more performant as well as lightweight ANNs (Whiteson et al. (2005)). However, it slows down the training process. Thus, *connection_density* default value is set to 1.0.

The training phase stops when one exit strategy is met. CSIP-train uses user-provided parameters *training_error* and *max_epochs* to enable three default exit strategies:

1. *"threshold error reached"* when the MSE of the best genome is smaller than *training_error*;
2. *"max epochs reached"* when neuroevolutionary algorithm reaches the number of *max_epochs* allowed;
3. *"constant error"* when the MSE of the best genome remains constant for 100 epochs.

In case *"normValid"* collection is available in the database, CSIP-train uses *"random validation"* dataset generated from TV dataset splitting phase to additionally provide overfitting exit strategy:

4. *"overtraining"* when the MSE of the best genome computed on *"random validation"* dataset starts increasing during the training.

If *"normValid"* collection is available in the database, CSIP-train uses this dataset to validate the SM; CSIP-train uses *"random validation"* dataset otherwise. Further details are provided in paragraph *PHASE 3: SM validation*.

With respect to overfitting issue, this phenomenon happens when a statistical model fits existing noise in the dataset instead of the underlying function (Razavi et al. (2012a)). This is a well known and studied phenomenon when it comes to applying statistical models to physical experiments and measured data.

In case of noise-free data obtained from deterministic simulation models runs (surrogate modeling applications), overfitting is still an issue, even if it is sometimes neglected (Sexton et al. (1998); Jin et al. (2002); Razavi et al. (2012a)). Here, overtraining mainly happens when the statistical model is overparameterized in regard to available dataset size (large degree of freedom).

The actual potential problem for surrogate modeling applications is *"conformability of the model structure with the shape of the available data"* (Razavi et al. (2012a)). Regression analysis based off of curve-fitting with predefined model structure are not affected by the *conformability* issue since the assigned shape of model structure covers the entire input space. Opposingly, ANN methodologies are highly influenced by *conformability issue* since their behavioural emulation of original model snapshots results from combination of several local flexible nonlinear unit responses. Razavi et al. (2012a) demonstrates the conformability issue with two single-hidden-layer ANNs: a more flexible structure with 15 neurons, and a more parsimonious structure with 8 hidden neurons. However, this constraint is mainly emphasized when the dataset is not well distributed over the domain space to properly describe the model behaviour.

Generally speaking, ANN methodologies have proven of high capability of properly recognizing and emulating underlying function in most application domain (Razavi et al. (2012a)). Nevertheless, Razavi et al. (2012a) suggests early stopping and Bayesian regularization to overcome ANN-based SM overfitting and conformability issues.

FeNS currently proposes four approaches to avoid ANN unpredictable fluctuations and fortifies the entire methodology:

1. early stopping, even though this methodology requires big dataset which is not always available;
2. ensemble system of SMs coupled to uncertainty quantification of eSM results, which smooth potential unpredictable behaviour of each single SM;
3. TV dataset random split based off of same probability distribution of the two outcoming datasets;
4. reduced degree of freedom by selecting relevant original model input parameter only (based off of scientist knowledge + FS-NEAT).

Further investigations on this problem are required since FS-NEAT is an innovative approach to environmental SM applications, and no background literature exists on the topic. Additionally, FS-NEAT notably creates highly flexible and formally unstructured ANNs, which behaviour in SM applications has to be deeply examined and tested. Supplementary investigations on automated integration of Halton sequence approach to design of experiments (DoE) to and Bayesian regularization are required as well.

Nonetheless, FeNS system has been currently tested on case studies prepared by attempting to homogeneously cover the entire input space.

PHASE 3: SM validation

This step executes during the training phase depending upon *recovery_epochs* user-defined parameter and when the training process is completed. Here, the partially or fully trained SM runs with *"random validation"* dataset, or *"common validation"* input dataset if available. SM estimates are compared to *"random validation"* or *"common validation"* original model results through a sequence of efficiency/goodness of fit indices (Figure 31):

- absDiff: absolute difference;
- absVolumeError: absolute volume error;
- dsGrad: double sum analysis gradient;

- Fhf: Fenicia high flow;
- Flf: Fenicia low flow;
- Ioa: index of agreement;
- Kge: Kling and Gupta efficiency;
- Modeldev: model deviation;
- Nashsutcliffe: Nash-Sutcliffe efficiency;
- Nbias: bias error;
- Pbias: percent bias;
- Personscorrelation: pearson correlation;
- Pwrmse: peak weighted root mean squared error;
- R2: r squared;
- Rmse: root mean squared error;
- transformeRmse: transformed root mean squared error.

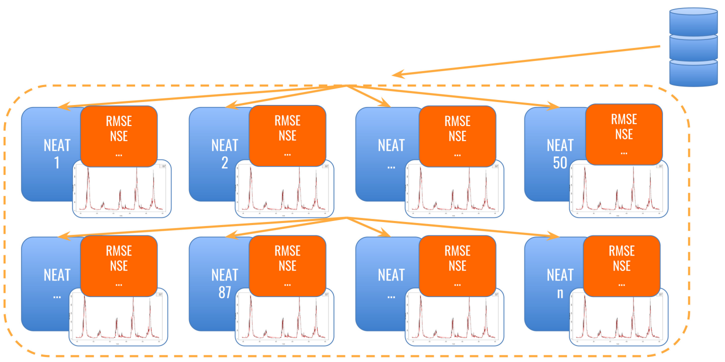


Figure 31: Conceptual approach of CSIP-validation service.

Since the NEAT algorithm creates the ANN from a layer of input nodes and a layer of output nodes, user might be interested in following growth and evolution of the ANN structure. As a result, the structure of the best genome is analyzed right after the validation phase and number of input nodes, output nodes, hidden nodes, and links becomes part of the ANN historical evolution metadata.

PHASE 4: SM store

This step executes after PHASE 3: SM validation during the training phase depending upon *recovery_epochs* user-defined parameter and when the training process is completed.

During this step, CSIP-train collects metadata and ANN structure and stores them into the database (database design is thoroughly analyzed in section *MongoDB: formal structure*). CSIP-train service implements MongoDB GridFS API to store the serialized structure of the SM. GridFS automatically stores the binary file of the SM in *"trained.chunk"* collection and metadata in *"trained.files"* collection. SM documents in *"trained.chunk"* collection and *"trained.files"* collection are overridden every time CSIP-train pushes SM information to the database. However, CSIP-train running service keeps track of best genome score and structure evolution during training and growing arrays of historical evolutions.

MongoDB: formal structure

Figure 32, Figure 33, and Figure 34 shows the formal structure of the *"trained.files"* collection. *"trained.chunk"* collection contains only the binary format of the SM structure. In addition to *_id*, *length*, *chunkSize*, *uploadDate*, and *md5* which are automatically generated by MongoDB, the BSON document of an SM contains a metadata BSON document. Here, *nn_id* and *suid* (Figure 32) are hooks for CSIP-train service to:

1. delete the currently stored SM BSON document when a newer document is ready to overwrite (SM store recovery phase);
2. connect to the running service to check its status;

respectively. *variables* document is a carry on of variables metadata from *"normalized"* collection (Figure 23). *hyper_params* document collects user-defined parameters in the JSON payload (Figure 33). *performance* document stores last computed goodness of fit (GoF) indices of the partially or fully trained SM per output (Figure 33). history document contains number of *epochs* and *exit_strategy* when the training is completed as well as the arraylist of scores (MSE) of the best genome over the training (Figure 34). This arraylist grows over number of pushed recovery information. *best_net_structure* stores the evolution of the structure of the best genome over the training (Figure 34).

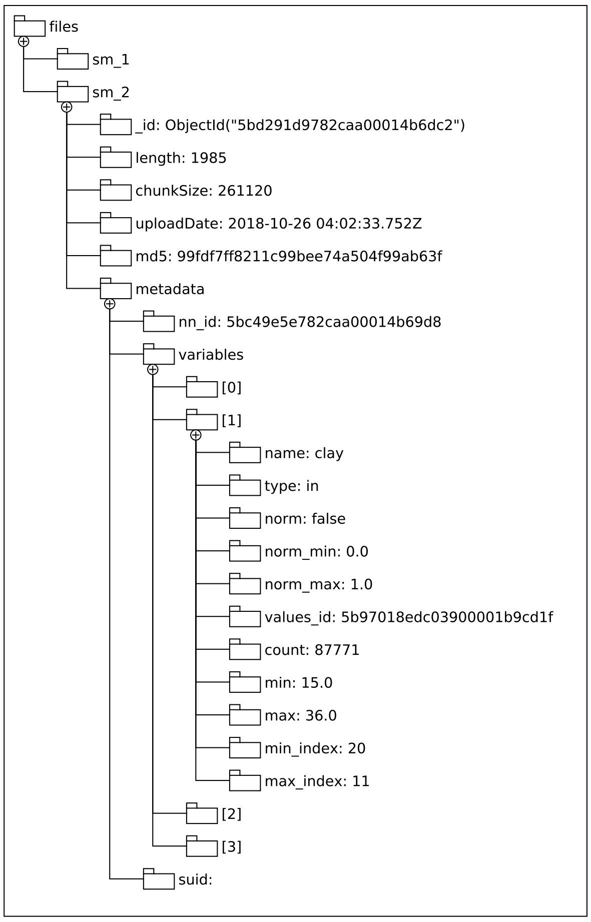
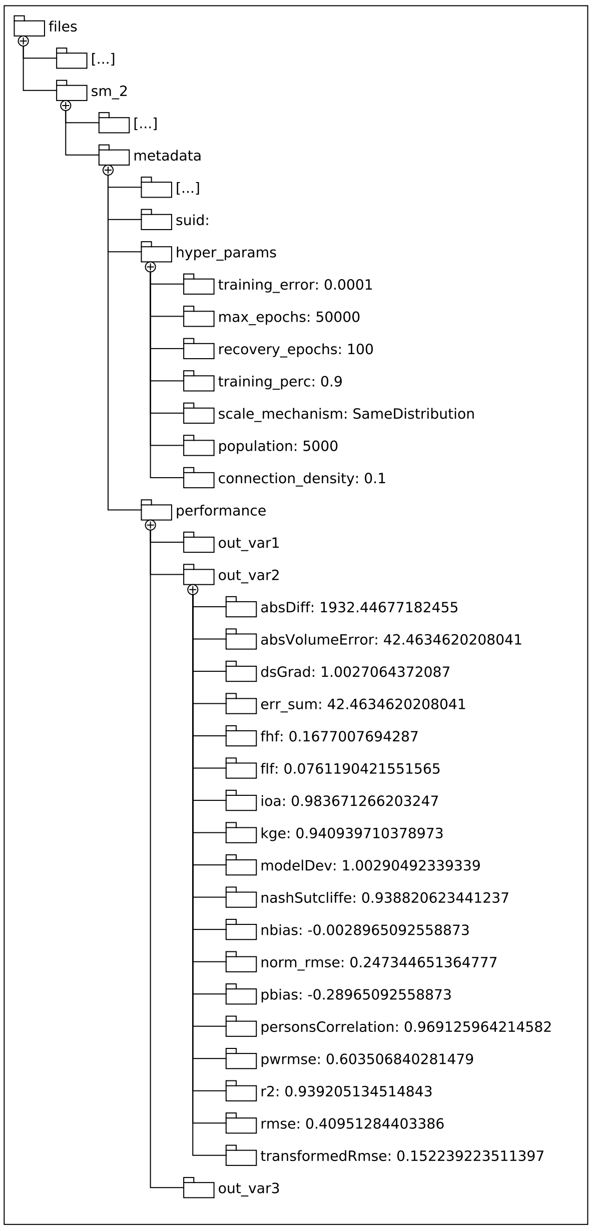


Figure 32: Formal structure of MongoDB *"trained.files"* collection



```
files
    [...]
    sm_2
        [...]
        metadata
            [...]
            suid:
            hyper_params
                training_error: 0.0001
                max_epochs: 50000
                recovery_epochs: 100
                training_perc: 0.9
                scale_mechanism: SameDistribution
                population: 5000
                connection_density: 0.1
            performance
                out_var1
                out_var2
                    absDiff: 1932.44677182455
                    absVolumeError: 42.4634620208041
                    dsGrad: 1.0027064372087
                    err_sum: 42.4634620208041
                    fhf: 0.1677007694287
                    flf: 0.0761190421551565
                    ioa: 0.983671266203247
                    kge: 0.940939710378973
                    modelDev: 1.00290492339339
                    nashSutcliffe: 0.938820623441237
                    nbias: -0.0028965092558873
                    norm_rmse: 0.247344651364777
                    pbias: -0.28965092558873
                    personsCorrelation: 0.969125964214582
                    pwrmse: 0.603506840281479
                    r2: 0.939205134514843
                    rmse: 0.40951284403386
                    transformedRmse: 0.152239223511397
                out_var3
```

Figure 33: Formal structure of MongoDB *"trained.files"* collection

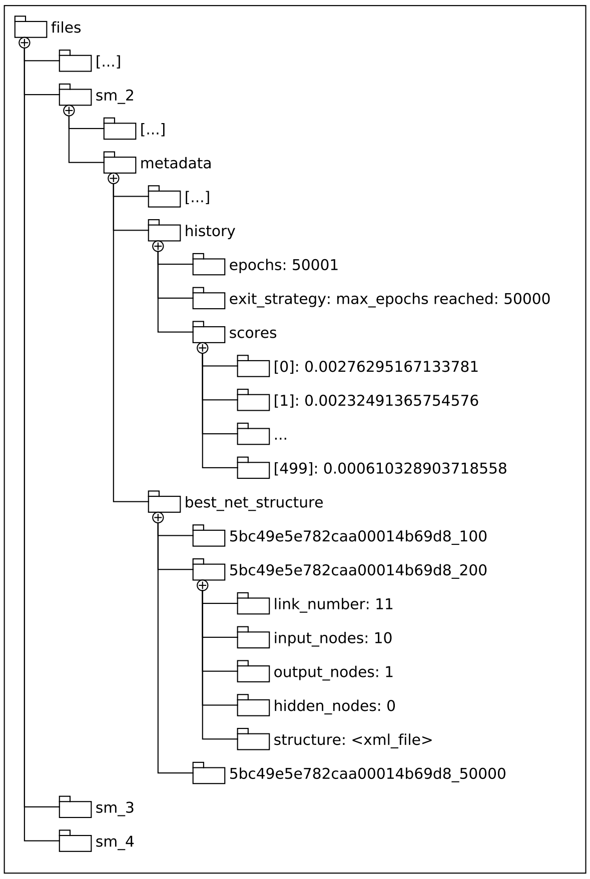


Figure 34: Formal structure of MongoDB *"trained.files"* collection

3.3.2.5.4 Service 4: SMs selection, building the ensemble

The fourth service goes through the performance of every trained and validated SM, picks the utmost performant ones based off of user defined criteria and stores SM IDs into a separated collection named *"selected"*.

SM might train to provide more than one answer (more than one output node). However, CSIP-select provides selection algorithms that check performance on a single output only.

CSIP-select service is called when the training phase of a collection of SMs is over and every SM has been validated against the validation dataset. Figure 35 shows the interactive behaviour of the CSIP-select service through a sequence diagram.

In addition to the application specific database name, the JSON payload contains four fields required to set up the selection mechanism: the output *variable* to check SM performance of, the *mechanism* type, the *threshold* value, and the *error* type (Listing 3.23).

Listing 3.23: Template JSON payload of CSIP-select service.

```
{
  "metainfo": {},
  "parameter" : [
    {
      "name": "annName",
      "value": "db_name"
    },
    {
        "name": "variable",
      "value": "out_var"
    },
    {
      "name": "mechanism",
      "value": "percentile"
    },
    {
      "name": "threshold",
      "value": 95
    },
    {
      "name": "error",
      "value": "nashSutcliffe"
    }
  ]
}
```

Here, the modeler may want to select all the available SMs or pick the utmost performant ones to create the ensemble of SMs.

Consequently, CSIP-select service provides three selection mechanisms:

1. *error* mechanism loops over user-choice statistical error of each trained SM and stores valid SMs and their IDs, which performances are above a user defined threshold, into two lists;

2. *percentile* mechanism (1) computes the probability distribution of user-choice statistical error of valid SMs, (2) identifies error threshold for user provided percentile, (3) loops over user-choice statistical error of each trained SM and stores valid SMs and their IDs, which performances are above the threshold, into two lists;

3. *number* mechanism simply selects the *n*-utmost performant SMs depending on the user chosen statistical error and stores valid SMs and their IDs into two lists.

CSIP-select selection mechanism identifies valid SMs by checking if *values_id* in each SM metadata (Figure 32) is identical to *values_id* in *"raw"* collection metadata (Figure 23): if *values_id* are identical, the SM has been trained on last provided dataset and is marked as selectable consequently; if *values_id* differs,

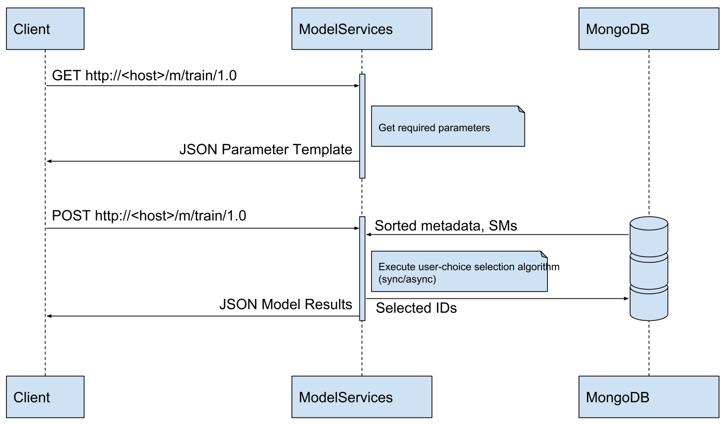


Figure 35: Sequence diagram of CSIP-select service.

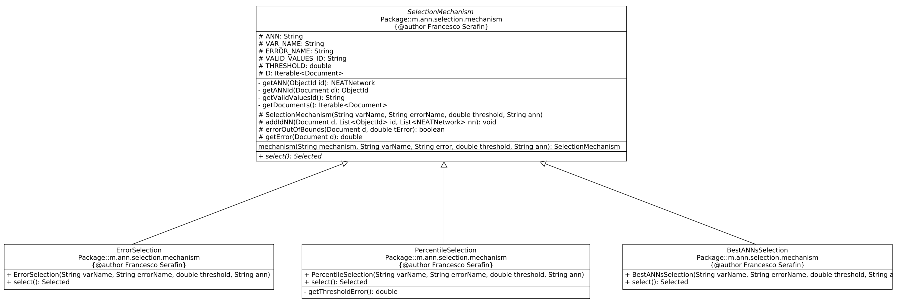

SelectionMechanism
Package::m.ann.selection.mechanism
{@author Francesco Serafin}
# ANN: String
# VAR_NAME: String
# ERROR_NAME: String
# VALID_VALUES_ID: String
# THRESHOLD: double
# D: Iterable<Document>
- getANN(ObjectId id): NEATNetwork
- getANNId(Document d): ObjectId
- getValidValuesId(): String
- getDocuments(): Iterable<Document>
# SelectionMechanism(String varName, String errorName, double threshold, String ann)
# addIdNN(Document d, List<ObjectId> id, List<NEATNetwork> nn): void
# errorOutOfBounds(Document d, double tError): boolean
# getError(Document d): double
mechanism(String mechanism, String varName, String error, double threshold, String ann): SelectionMechanism
+ select(): Selected
ErrorSelection
Package::m.ann.selection.mechanism
{@author Francesco Serafin}
+ ErrorSelection(String varName, String errorName, double threshold, String ann)
+ select(): Selected
PercentileSelection
Package::m.ann.selection.mechanism
{@author Francesco Serafin}
+ PercentileSelection(String varName, String errorName, double threshold, String ann)
+ select(): Selected
- getThresholdError(): double
BestANNsSelection
Package::m.ann.selection.mechanism
{@author Francesco Serafin}
+ BestANNsSelection(String varName, String errorName, double threshold, String a
+ select(): Selected

Figure 36: UML of implemented selection mechanism.

the SM has been trained on a older and thus partial dataset and is marked as unselectable consequently.

CSIP-select selection mechanism retrieves already sorted SMs and their metadata, from the most to the less performant one. This is achieved by pushing to the database a sorting algorithm properly tuned depending on positive or negative trend of user-choice statistical error to process. This architectural choice has four advantages:

1. speeds up the sorting process by leveraging MongoDB native algorithms;
2. avoids the burden of designing effective sorting algorithm service-side and attempts to delegate as many operations as possible database-side;
3. shortens selection loop since SMs that perform worse than user-choice threshold are the last in the processing list and never checked consequently;
4. invokes one single sorted information transfer from database to running service.

Selection mechanism is designed implementing the Simple Factory principle through a static method. Additional selection mechanism are easily implemented by extending `SelectionMechanism` abstract class and overriding the `select` method (Figure 36).

The ensemble of SMs is created once the utmost performant SMs are selected. To attempt to estimate ensemble accuracy, CSIP-select runs the ensemble of SMs against available dataset, which is:

A. *common validation* dataset if available;

B. *TV dataset* otherwise,

uncertainty quantifies ensemble estimate, and compute percentage of original model results between quartiles and min-max.

This is not an optimal design choice since it doesn't properly describe the ensemble of SMs accuracy. More investigation is required.

Finally the IDs of the selected SMs are stored into *"selected"* collection along with ensemble metadata.

MongoDB: formal structure

Figure 37 shows the formal structure of the *"selected"* collection. In addition to *_id* and *timestamp*, which are provided by default, *selected_id* contains the array of selected SM IDs, while *percentage_btw_quartiles* and *percentage_btw_min-max* store information of ensemble of SMs accuracy.

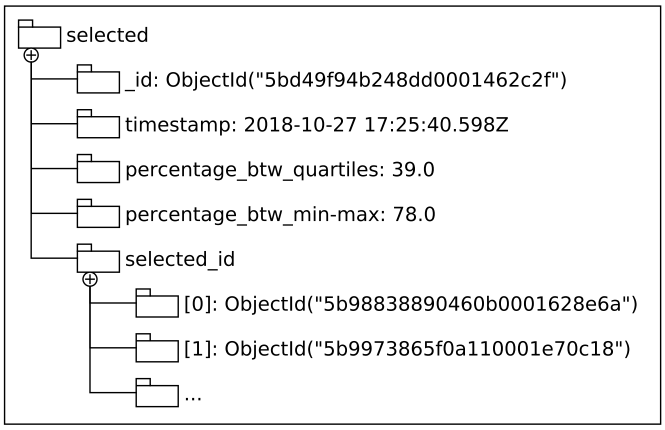

selected
_id: ObjectId("5bd49f94b248dd0001462c2f")
timestamp: 2018-10-27 17:25:40.598Z
percentage_btw_quartiles: 39.0
percentage_btw_min-max: 78.0
selected_id
[0]: ObjectId("5b98838890460b0001628e6a")
[1]: ObjectId("5b9973865f0a110001e70c18")
...

Figure 37: Formal structure of MongoDB *"selected"* collection.

3.3.2.5.5 Service 5: eSM run

The fifth service runs the ensemble of selected SMs against user provided data. Figure 38 shows the interactive behaviour of the CSIP-run service through a sequence diagram.

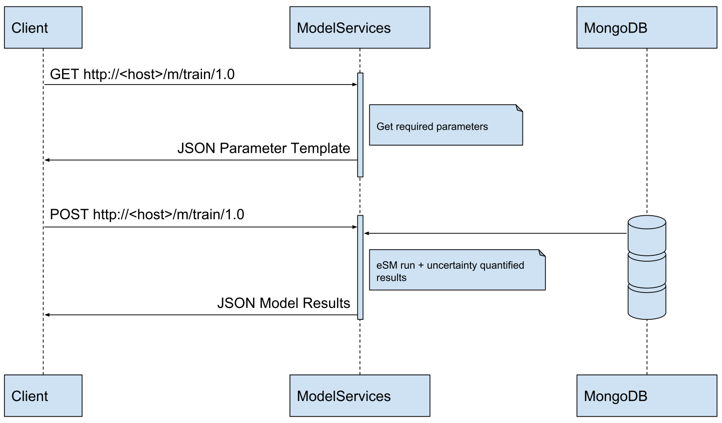


Figure 38: Sequence diagram of CSIP-run service.

CSIP-run service is the only exposed service to on-the-field personnel. It just requires the input snapshot to provide uncertainty quantified result. Consequently, the JSON payload is a simplified version of CSIP-collect payload (Listing 3.12): no output parameters need to be provided and input parameters require no description since their required metadata are stored in the database already (Listing 3.24).

Listing 3.24: Template JSON payload of CSIP-run service.

```
{
    "metainfo": {},
    "parameter": [
        {
          "name": "annName",
          "value": "db_name"
        },
        {
          "name": "in_var1",
          "value": 0.1
        },
        {
          "name": "in_var2",
          "value": 19
        },
        {
          "name": "in_var3",
          "value": 5
        },
        {
          "name": "in_var4",
          "value": 0
        },
        {
          "name": "in_var5",
          "value": 75
        }
    ]
}
```

After parsing the JSON payload, CSIP-run service retrieves metadata from *"normalized"* collection to normalize user-provided snapshot, retrieves and run the ensemble of SMs, and returns denormalized uncertainty quantified results (min, first quartile, median, third quartile, and max). The JSON response may look like Listing 3.25.

Listing 3.25: Generic JSON response of CSIP-run service.

```
{
  "metainfo": {
    "status": "Finished",
    "suid": "221c7df8-da0e-11e8-8b41-0f54b71099b1",
    "cloud_node": "10.43.0.16",
    "request_ip": "129.82.52.206",
    "service_url": "http:\/\/csip.engr.colostate.edu:8088" +
                   "\/csip-ann\/m\/run\/1.0",
    "tstamp": "2018-10-27 11:31:27",
    "cpu_time": 35,
    "expiration_date": "2018-10-27 11:31:57"
  },
  "parameter": [
    {
      "name": "annName",
      "value": "8088-r2_bv_ch_cl_wr_53_110_he"
    },
    {
      "name": "slope",
      "value": 14
    },
    {
      "name": "length",
      "value": 80
    },
    {
      "name": "stir",
      "value": 110
    },
    {
      "name": "contour",
      "value": 0
    },
    {
      "name": "kffact",
      "value": 0.37
    },
    {
      "name": "sand",
      "value": 5
    },
    {
      "name": "silt",
      "value": 63
    },
    {
      "name": "clay",
      "value": 32
    },
    {
      "name": "biomass",
      "value": 105.187140718212
    },
    {
      "name": "r_factor",
      "value": 140.549
    }
  ],
  "result": [{
    "name": "erosion",
    "value": 18.717869790589237,
    "min": 18.210253820784978,
    "1q": 18.54484936734709,
    "3q": 18.94649079819633,
    "max": 19.21732651235383,
    "percentage btw quartiles": 23,
    "percentage btw min-max": 55,
    "vals": [
      19.087608233884957,
      18.936707790276103,
      19.21732651235383,
      18.64971025814681,
      18.52709377865761,
      18.786029323031663,
      18.210253820784978,
      18.59811613341554,
      18.523385448879388,
      18.949751800836403
    ]
  }]
}
```

3.4 CASE STUDIES

This section introduces to actual applications of the FeNS methodology. To demonstrate its goodness, the following experiments are carried out on real test cases.

The entire experiment suite is focused on dimensionality reduction. In order to facilitate as much as possible the access to original model knowledge, the number of input resources required for an eSM run was kept as lowest as possible. Scientist expertise allowed for selecting only the indispensable parameters or estimate weighted averaged values representative of a specific phenomenon.

Every experiment carried out followed this structure:

1. Data collection: NEAT-based SM is data driven. This means that the SM learns the mathematical model behaviour from the analysis of a large variety of input/output snapshots. In order to generate the training/testing dataset

for each specific experiment, the actual mathematical model run several times under different scenarios. The model involved in this suite of experiments is hosted as CSIP services at CSU super-computing environment.

2. Experiment run: this step involves the chain of FeNS services for creating and running the ensemble of surrogate models. The suite of experiments allowed for debugging and refining web-services pipeline.

3. Result analysis: boxplots are generated out of uncertainty quantified results from eSM run against testing dataset. Further error analysis is performed.

4. Following DoE: in order to generalize the SM behaviour by emulating a broader variety of scenarios, more complex DoE are designed and only indispensable additional parameters are selected.

3.4.1 RUSLE2

Version 2 of the Revised Universal Soil Loss Equation (RUSLE2) is a mathematical model that allows for estimating soil loss, sediment yield, and sediment characteristics as a result of rill and interrill erosion phenomena generated by rainfall and related runoff (Foster (2005)). RUSLE2 can be applied to large scale analysis for erosion rate inventory, or "field" scale geographic areas to estimate potential erosion rates for guiding conservation and erosion control planning. RUSLE2 is usually applied on a large variety of land use: cropland, pastureland, rangeland, disturbed forestland, construction sites, mined land, reclaimed land, landfills, military lands (Foster (2005)).

DoE for to the creation of SMs for RUSLE2 are mainly focused on emulating the soil erosion as a results of land management for crop rotation corn and soybeans in Iowa.

The main reason relates to the importance of corn: at a global scale, it is the most valuable grain crop (used for human food, livestock feed, and biofuel) and the USA itself produces over 36% (Green et al. (2018a)). This crop is mainly grown within the Midwest Corn Belt, which includes 12 Midwest states (from East to West, Figure 39): Ohio, Kentucky, Michigan, Indiana, Illinois, Wisconsin, Missouri, Iowa, Minnesota, Kansas, Nebraska, and South Dakota.

This DoE focuses on scenarios in Iowa because it is the main producer of corn along with Illinois and Minnesota. Thus, consultant and planning agencies like USDA-NRCS are really focused and interested in SMs able to emulate RUSLE2 behaviour for Iowa scenarios.

3.4.1.1 *DoE 1*

The first experiment aimed to test the capability of FeNS system to recognize and emulate the hidden nonlinear function that describes RUSLE2 behaviour with respect to predicted soil erosion.

Additionally, the first experiment served as a guinea pig to test that selected input parameters were actually adequate to describe model behaviour for a simplified scenario.

3.4.1.1.1 DoE 1 - Step 1

The first step of the DoE was focused on generating the training/testing dataset for the following scenario:

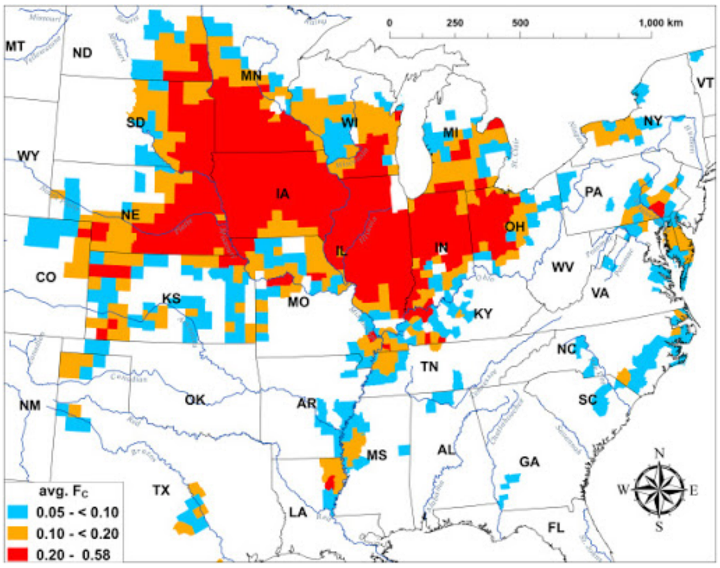


Figure 39: Geographic distribution of the Corn Belt, credit (Green et al. (2018a)).

- **Location:** Cherokee county (IA). In this area (Northwest Iowa), Loess is the predominant silt-sized sediment. Galva-Primghar are the major soil components. Slopes are mostly gentle (or nearly level), but can get very steep bordering stream valleys. This is a great first test case in order to vary from low to high steepness which returns low and high soil erosion values (Figure 40);

- **Field length:** constant (100 m);

- **Field steepness:** varying by 0.1% in between admissible soil optimal range;

- **Field contouring:** simulation runs with contoured fields or not;

- **Land management:** 8 types of soil managements. In order to reduce the list of operations and management practices to a single value per type, the soil tillage intensity rate (STIR) of each land operation have been summed into a single value;

- **Soil:** 4 types of soils of the same Galva family. Every soil is described by a number of parameters which result from surveys collected into SSURGO database. Three parameters were selected to represent soil behaviour: KF Factor (soil erodibility), Silt percentage, and Component percentage (the percentage of that type of soil in a soil sample).

R2:8088 v2.1 CSIP-service was concurrently hit around 500 times, properly permuting previously listed parameters. This process was automated by developing a proper Python3 script leveraging to:

1. read in an input JSON template payload;

2. replace standard parameters with properly permuted values;

3. run the CSIP-service;

4. parse the output json payload and store erosion results.

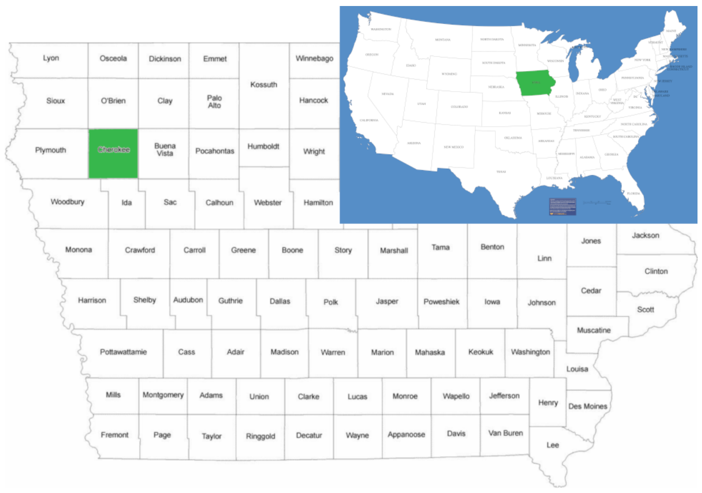


Figure 40: Cherokee county in Iowa.

3.4.1.1.2 DoE 1 - Step 2

The second step started by shuffling collected data and splitting it into 90% TV dataset and 10% testing. The TV dataset was uploaded onto a new MongoDB database hosted on erams10 server. After normalization process, only one surrogate model was created for the sake of testing. The final structure of the artificial neural network had 6 input nodes (Figure 41):

- Steepness
- Contouring (yes, 1, or no, 0)
- STIR value
- KF factor
- Silt percentage
- Component percentage

The only output node was the erosion rate.

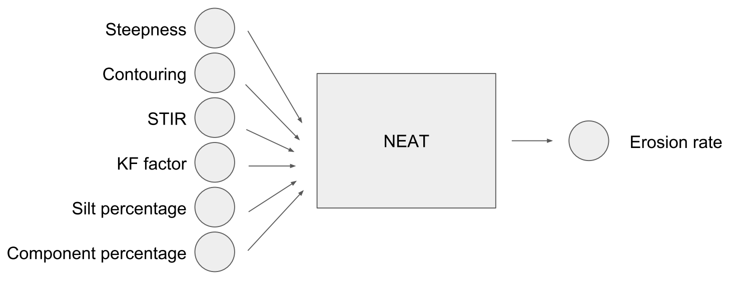


Figure 41: Generic SM input/output structure for DoE 1.

The surrogate model was generated using default parameters:

- Initial population = 1 000
- Connection density = 1
- Final training error = 10^{-4}

- Training data set = 90%
- Validation data set = 10%
- Splitting mechanism = "Same Distribution"

3.4.1.1.3 DoE 1 - Step 3

Consequently, the testing dataset run against the surrogate model. SM results are plot against R2 original model results and analyzed.

Plot in Figure 42 and Figure 43 illustrate results comparison. Figure 42 shows the actual comparison between observed (origin model runs are represented with a black circle) and simulated (SM estimated results are represented with a red cross) values. Preliminary results show that the single ANN was able to learn and accurately replicate RUSLE2 behaviour.

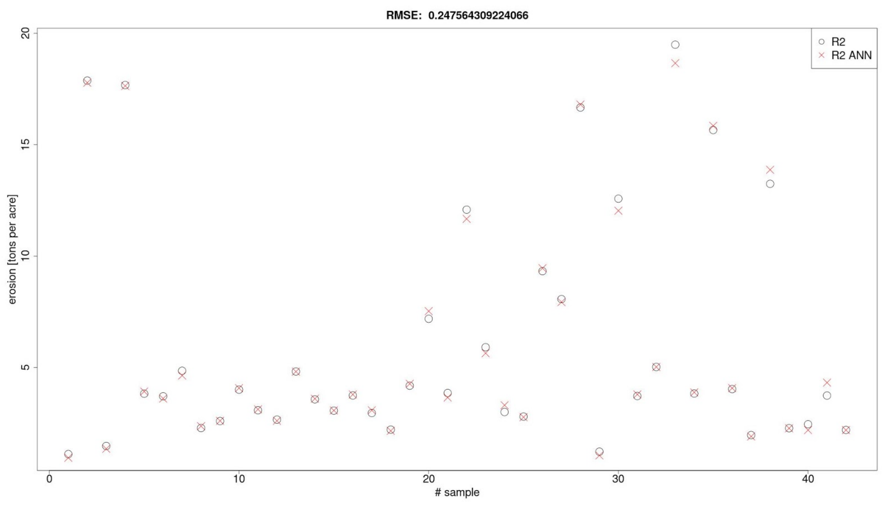


Figure 42: Red crosses represents SM estimates, while black dots represents original RUSLE2 runs.

Figure 43 illustrates the scatter-plot of observed vs simulated. Here, most of the erosion values lay on the 1:1 line or are really close, which means that accurate results are predicted for low and high erosion values. Even if high erosion values are rarely generated from model runs, the SM is generally capable of emulating the original model behaviour.

To conclude this preliminary test, the SM is able to understand and learn RUSLE2 behaviour. Selected input parameters sufficiently replicate/describe original model behaviour and drive SM learning to accurately predict erosion rates. However, component percentage employed in this experiment is not a real soil parameter and can potentially mislead the learning process.

3.4.1.1.4 DoE 1 - Step 4

This introductory test case supported development and validation of the FeNS methodology. It was an important step to assess applicability of NEAT algorithms to surrogate modeling practices. Furthermore, this introductory test case resulted in first exercise of NEAT capabilities applied to an environmental topic, which is relevant since it has never done before. This was an important test case to start debugging the entire FeNS pipeline, since the dataset was easy to generate and manage.

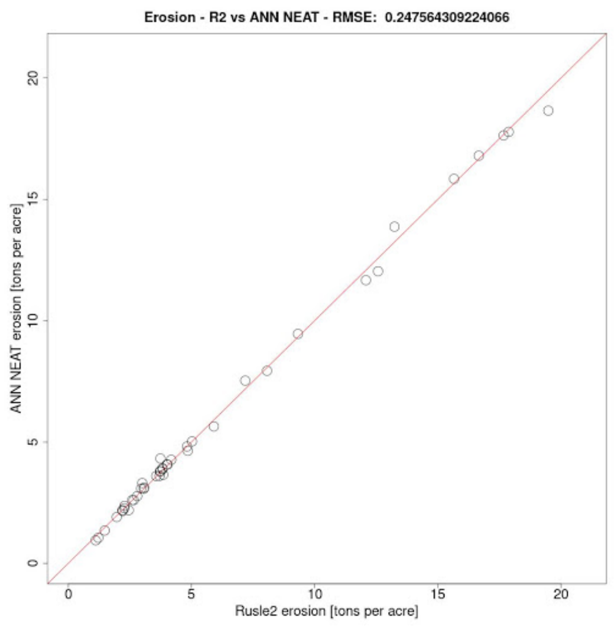


Figure 43: Scatterplot of SM estimates against RUSLE2 results.

Nevertheless, this first DoE was a simple example. DoE 2 was driven by the need of increasing the range of values of input parameters. 2 additional soil managements and 15 soil types from different soil families were included in the dataset generation. Furthermore, the study area was expanded from a single county to two counties to account for different weather conditions.

3.4.1.2 *DoE 2*

The second experiment aimed to confirm NEAT capabilities to recognize, learn and emulate the hidden nonlinear function that describes RUSLE2 behaviour with respect to predicted soil erosion.

Compared to DoE 1, a larger range of values of relevant input parameters was used. 15 additional soil types and 2 more soil managements became part of the dataset to generate a broader variety of RUSLE2 runs.

This experiment was conducted excluding component percentage, a previously used input parameter, since it doesn't really describe any actual soil characteristic and might potentially mislead training process and SM emulation capabilities. Opposingly, clay and sand percentages became part of the learning dataset to comprehensively describe soil properties. To account for different weather conditions, the study area was extended from one to two counties in the State of Iowa.

In order to fasten the training process, the dataset was split into two separate data clusters based upon STIR value (input parameter):

- Cluster1 : STIR 5 →53
- Cluster2 : STIR 53 →110

where STIR = 53 is the overlapping boundary.

The final overall SM structure result in a *modular neural network*: the input domain is split into multiple sub-domains, each sub-domain is assigned to a responsible expert module. Eventually, a complex mapping problem is decomposed into several simpler ones.

As a result, two separate ensemble of SMs with identical structure but different responsibility were generated.

Summarizing, one input node was removed (*component percentage*) and three new input nodes were added (*clay percentage*, *sand percentage*, and *weather condition*) to the final structure of the SM.

3.4.1.2.1 DoE 2 - Step 1

The first step of the DoE was focused on generating the training/testing dataset for the following scenario:

- **Location:** Buena Vista and Clay counties (IA). This area is still northwest Iowa. Here, loess with silt-sized sediment and glacial till with unsorted glacial sediment are the predominant soils. Slopes are nearly level to moderately sloping. This test case allows for taking into account two different weather conditions (Figure 44);
- **Field length:** constant (100 m);
- **Field steepness:** varying by 0.1% in between admissible soil optimal range;
- **Field contouring:** simulation runs with contoured fields or not;
- **Land management:** 10 types of soil managements. In order to reduce the list of operations and management practices to a single value per type, the soil tillage intensity rate (STIR) of each land operation have been summed into a single value;
- **Soil:** 19 types of soils from different families. Every soil is described by a number of parameters which result from surveys collected into SSURGO database. Four parameters were selected to represent soil behaviour: KF Factor (soil erodibility), Silt, Clay, and Sand percentage.

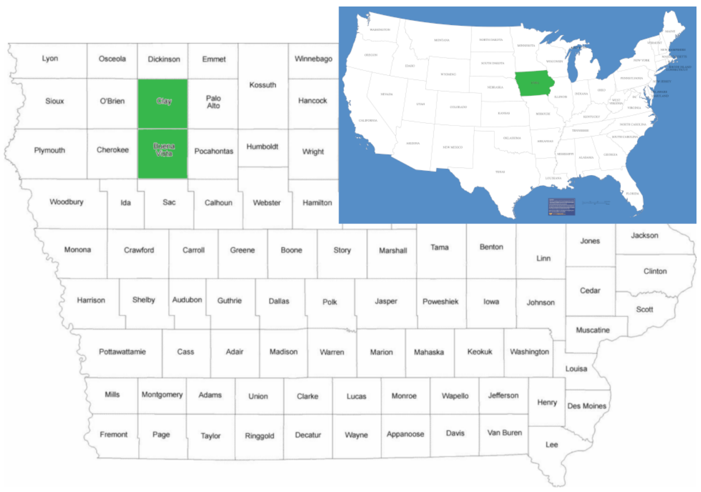


Figure 44: Buena Vista and Clay counties in Iowa.

R2:8088 v2.1 CSIP-service was concurrently hit about 5000 time, properly permuting previously listed parameters. This process was automated by developing a proper Python3 script leveraging to:

1. read in an input JSON template payload;

2. replace standard parameters with properly permuted values;
3. run the CSIP-service;
4. parse the output json payload and store erosion results.

3.4.1.2.2 DoE 2 - Step 2

The second step started by shuffling collected data and splitting it into 90% TV dataset and 10% testing. The TV dataset was uploaded onto a new MongoDB database hosted on erams10 server. After normalization process, 10 SM were collected to generate an eSM per cluster. The final structure of each SM had 8 input nodes (Figure 45):

- R factor (weather condition)
- Steepness
- Contouring (yes, 1, or no, 0)
- STIR value
- KF factor
- Silt percentage
- Sand percentage
- Clay percentage

The only output node was the erosion rate.

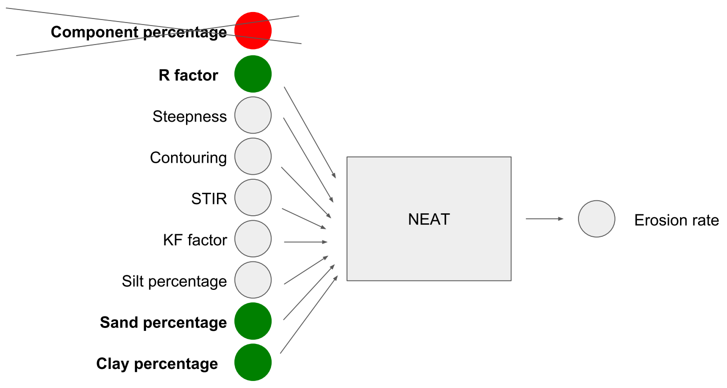


Figure 45: Generic SM input/output structure for DoE2.

The overall structure of the clustered eSM is shown in Figure 46. Here, the expert module returns original model emulated results based upon user-provided input parameters. If STIR value is less than 53, the first expert module returns uncertainty quantified results off of eSM cluster 1. If STIR value is greater than 53, the second expert module returns uncertainty quantified results off of eSM cluster 2.

If user question lays right on the cluster boundary (STIR value equal to 53), both expert modules answer the question. Consequently, FeNS system computes uncertainty quantified results off of the 20 collected answers: eSM cluster 1 and eSM cluster 2 together.
Each SM was generated using the following hyper parameters:

- Initial population = 5000

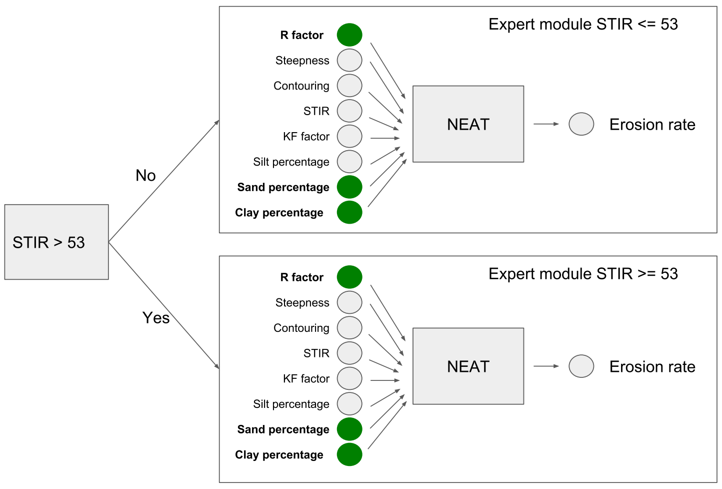


Figure 46: Expert modules design for DoE2.

- Connection density = 0.1
- Final training error = 10^{-4}
- Training dataset = 90%
- Validation dataset = 10%
- Splitting mechanism = "Same Distribution"

3.4.1.2.3 DoE 2 - Step 3

Consequently, the testing dataset run against the clustered eSM. SM estimate are plotted against RUSLE2 original model results and analyzed.

Figure 47 and Figure 48 show performance of first eSM cluster. Figure 47 shows the actual comparison between observed (original model runs are represented with a red cross) and eSM emulated uncertainty quantified (boxplots represent eSM runs) values. eSM estimates are pretty accurate since NS efficiency is above 0.99, RMSE is 0.13 t/acre and BIAS is only slightly negative -0.0069. The accuracy is mirrored in the scatterplot estimated vs original model erosion laying on a 1:1 line.

Figure 49 and Figure 50 show performance of second eSM cluster. Figure 49 shows the actual comparison between observed (original model runs are represented with a red cross) and eSM emulated uncertainty quantified (boxplots represent eSM runs) values. eSM estimates are pretty accurate since NS is above 0.98, RMSE is 0.3 t/acre and BIAS is only slightly positive 0.021. The accuracy is mirrored in the scatterplot estimated vs original model erosion laying on a 1:1 line.

3.4.1.2.4 DoE 2 - Step 4

DoE 2 confirmed goodness of NEAT methodology for surrogate modeling purposes. NEAT is still capable of learning and emulating original model behaviour even with enlarged dataset. FeNS system coupled to input data clustering allow for accurately answer user specific questions.

Nevertheless, DoE 2 still doesn't account for different field lengths. Additionally, a variety of crop yields need to be considered, since RUSLE2 provides erosion estimates based off of user-selected crop yield.

As a result, DoE 3 has been designed by including a larger study area, different field lengths, and varying crop yield.

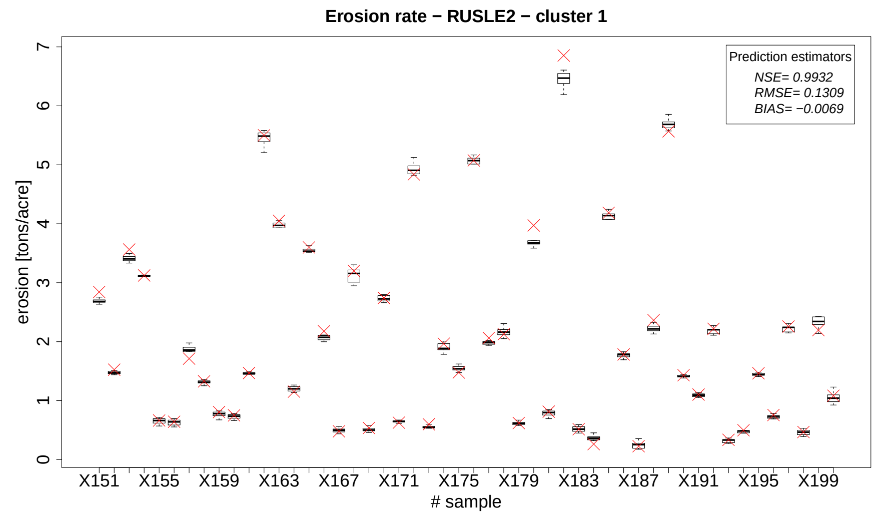


Figure 47: Cluster 1. Boxplots represents the ensemble of SMs estimates against RUSLE2 erosion runs (red crosses).

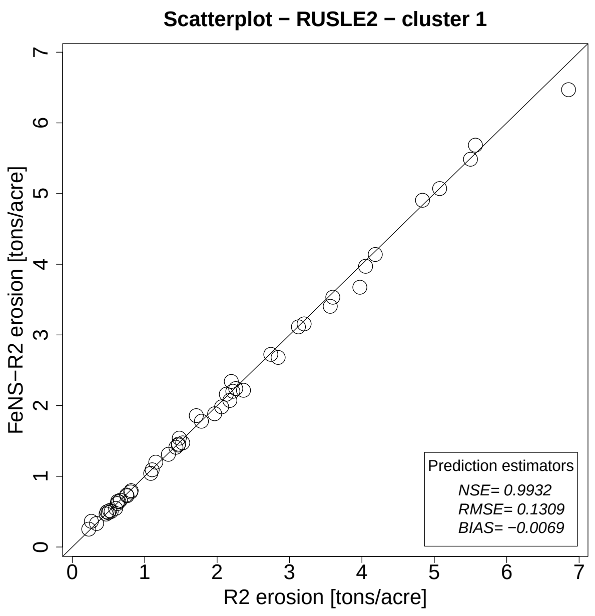


Figure 48: Cluster 1. Scatterplot of SM estimates (computed on the median of each boxplot) and RUSLE2 simulated values.

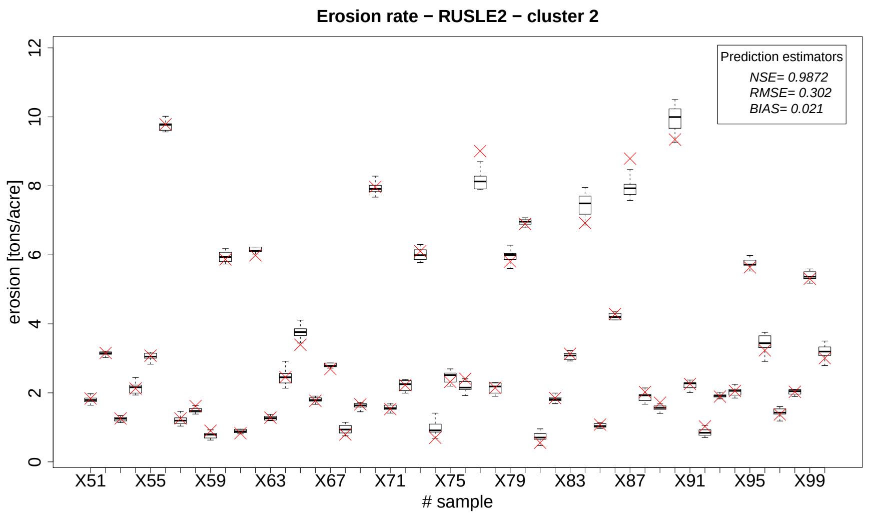


Figure 49: Cluster 2. Boxplots represents the ensemble of SMs estimates against RUSLE2 erosion runs (red crosses).

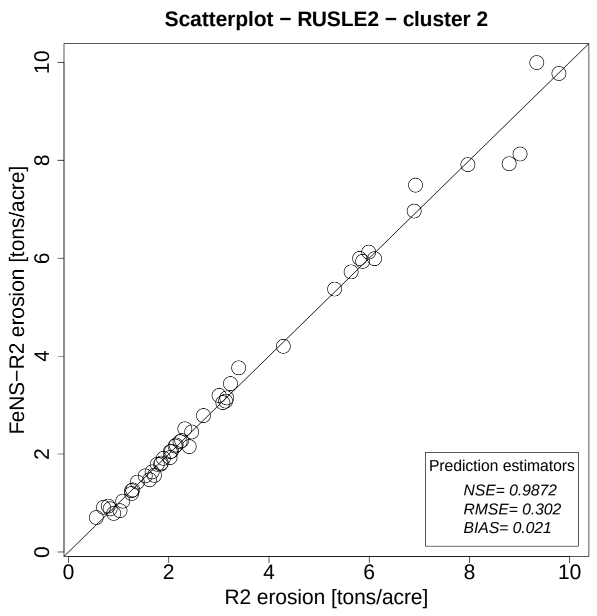


Figure 50: Cluster 2. Scatterplot of SM estimates (computed on the median of each boxplot) and RUSLE2 simulated values.

3.4.1.3 *DoE 3*

The third experiment aimed to account for a larger input space by adding two more input parameters, and by strategizing input space clustering in order to accurately estimate high erosion values.

The entire simulation results in a larger range of selected input parameters compared to DoE 2. 17 more soil types were used to generate a broad variety of RUSLE2 runs. Field length and weighted average crop rotation biomass were added in order to comprehensively describe field geometry and forecasted crop growth respectively. In order to account for different weather conditions, study area was expand from two to four different counties.

In addition to DoE 2 input data clustering on STIR value equal 53, dataset for STIR value greater than 53 was clustered on erosion value. Here, high erosion rates result from combination of high field steepness, heavily impacting soil management practices, and easily erodible soil. However, high erosion rates are rarely generated from RUSLE2 runs. Consequently, a dedicated cluster allows for properly emulating this localized specific model behaviour. As a result, the entire dataset was properly split into three clusters:

- Cluster1 : STIR 5 $\rightarrow$53
- Cluster2 : STIR 53 $\rightarrow$110 and erosion $<$ 11 tons/acre
- Cluster3 : STIR 53 $\rightarrow$110 and erosion $>$ 10 tons/acre

The three clusters have overlapping training dataset on STIR 53. The two clusters for $53 \leq$ STIR ≤ 110 have overlapping training dataset for erosion between 10 and 11 tons/acre. This makes both expert module able to answer question when the gate neural network foresees erosion bigger than 10 tons/acre and smaller than 11 tons/acre.

The final overall SM structure result in a **modular neural network**: the input domain is split into multiple sub-domains, each sub-domain is assigned to a responsible expert module. Eventually, a complex mapping problem is decomposed into several simpler ones.

As a result, three separate ensemble of SMs with identical structure but different responsibility were generated.

Summarizing, two new input nodes were added (field length, and weighted average of yield per crop) to the final structure of the SM.

3.4.1.3.1 DoE 3 - Step 1

The first step of the DoE was focused on generating the training/testing dataset for the following scenario:

- **Location:** Buena Vista, Cherokee, Clay, and Wright counties (IA). This area is still northwest Iowa. Four families of soil characterize the region involved: Northwest Iowa Loess, Tazewell Glacial Till, Loamy Wisconsin Glacial Till, and Clayey Lake Deposits. Here, loess with silt-sized erosional sediments, loamy glacial till, glacial outwash, and local alluvium are the predominant soils. Minor areas are covered in silty and clayey glacial lacustrine sediments overlying calcareous loamy glacial till. Slopes are nearly level to moderately sloping. Minor areas consist of broad, plane and convex ridges, long, convex side slopes, and concave drainageway. This test case allows for taking into account four different weather conditions (Figure 51);

- **Field length:** varying in between admissible steepness range;
- **Field steepness:** varying by 0.1% in between admissible soil optimal range;
- **Field contouring:** simulation runs with contoured fields or not;
- **Land management:** 10 types of soil managements. In order to reduce the list of operations and management practices to a single value per type, the soil tillage intensity rate (STIR) of each land operation have been summed into a single value;
- **Biomass:** weighted average of yield per crop. This parameter results from an average of crop characteristics in order to account for different corn and soybeans yields;
- **Soil:** 36 types of soils from different families. Every soil is described by a number of parameters which result from surveys collected into SSURGO database. Four parameters were selected to represent soil behaviour: KF Factor (soil erodibility), Silt, Clay, and Sand percentage.

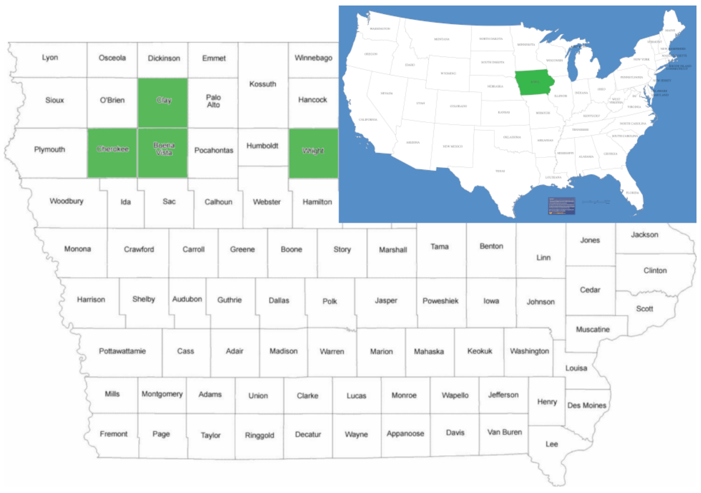


Figure 51: Buena Vista, Cherokee, Clay and Wrigth counties in Iowa.

R2:8088 v2.1 CSIP-service was concurrently hit about 180 000 time, properly permuting previously listed parameters. This process was automated by developing a proper Python3 script to:

1. read in an input JSON template payload;
2. replace standard parameters with properly permuted values;
3. run the CSIP-service;
4. parse the output json payload and store erosion results.

3.4.1.3.2 DoE 3 - Step 2

The second step started by splitting the entire dataset in three subsets: A) STIR $= 53$, B) STIR < 53, and C) STIR > 53. Afterwords collected data are shuffled and split it into 90% TV dataset and 10% testing. The TV dataset per cluster was uploaded in a new MongoDB database hosted on erams10 server. After normalization process, 10 surrogate models were collected to generate an eSM per cluster. The final structure of each artificial neural network had 10 input nodes (Figure 52):

- R factor (weather condition)
- Steepness
- Contouring (yes, 1, or no, 0)
- STIR value
- KF factor
- Silt percentage
- Sand percentage
- Clay percentage
- Biomass

The only output node was the erosion rate.

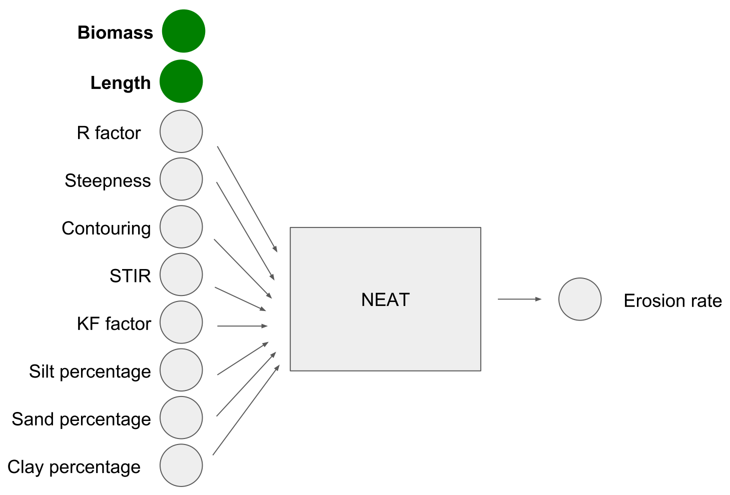


Figure 52: Generic SM input/output structure for DoE3.

The overall structure of the clustered eSM is shown in Figure 53. Here, the expert module returns original model emulated results based upon user-provided input parameters. If STIR value is less than 53, the first expert module returns uncertainty quantified results off of eSM cluster 1. If STIR value is greater than 53, the ANN gate is responsible of enabling the second or third expert module depending on its forecast of high or low erosion value. The ANN gate is a NEAT-generated ANN trained to forecast high or low erosion value based off of user provided input data. If the gate forecasts low erosion rate, the eSM cluster 2 is enabled. If the gate forecasts high erosion rate, the eSM cluster 3 is enabled (Figure 53).
Each SM was generated using the following hyper parameters:

- Initial population = 5 000
- Connection density = 0.1
- Final training error = 10^{-4}
- Training dataset = 90%
- Validation dataset = 10%
- Splitting mechanism = "Same Distribution"

Consequently, three ensembles of ANNs and a gating network were trained.

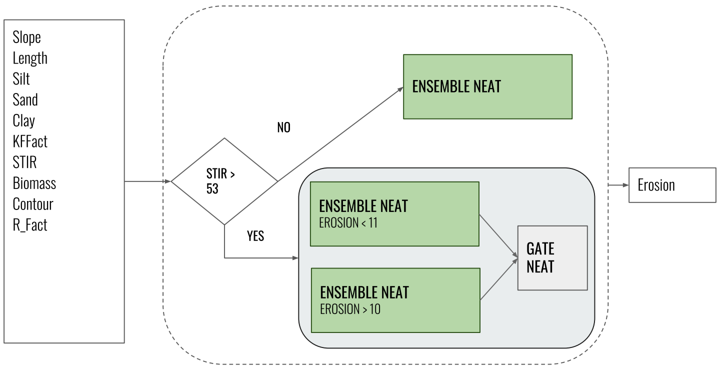


Figure 53: Expert modules design for DoE3.

3.4.1.3.3 DoE 3 - Step 3

Consequently, the testing dataset run against the clustered eSM. SM estimate are plotted against RUSLE2 original model results and analyzed.

Cluster STIR $\leq$ 53

Figure 54 and Figure 55 show performance of first eSM cluster. Figure 54 shows the actual comparison between observed (original model runs are represented with a red cross) and eSM emulated uncertainty quantified (boxplots represent eSM runs) values. eSM estimates are pretty accurate since NS efficiency is above 0.95, RMSE is 0.3297 and BIAS 0.0033. The accuracy is mirrored in the scatterplot where estimated vs original model erosion values lay on a 1:1 line.

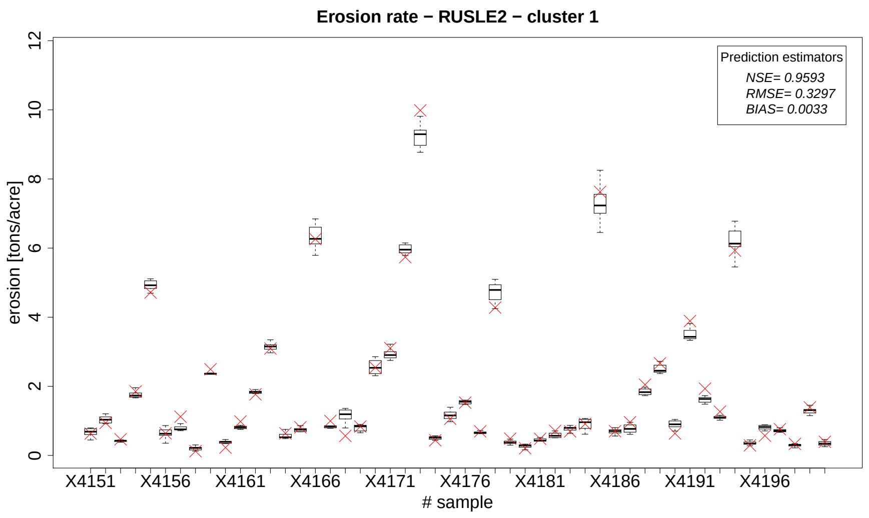


Figure 54: Cluster 1. Boxplots represents the ensemble of SMs estimates against RUSLE2 erosion runs (red crosses).

Cluster STIR $\geq$ 53 and erosion $<$ 11 tons/acre

Figure 56 and Figure 57 show performance of second eSM cluster. Figure 56 shows the actual comparison between observed (original model runs are represented with a red cross) and eSM emulated uncertainty quantified (boxplots represent eSM runs) values. eSM estimates are pretty accurate since NS efficiency is above

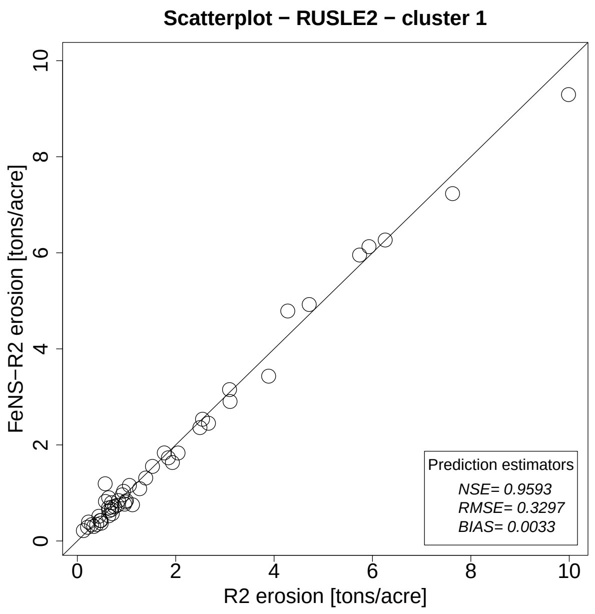


Figure 55: Cluster 1. Scatterplot of SM estimates (computed on the median of each boxplot) and RUSLE2 simulated values.

0.97, RMSE is 0.32 and BIAS is 0.0116. The accuracy is mirrored in estimated vs original model erosion values laying on a 1:1 line in the scatterplot.

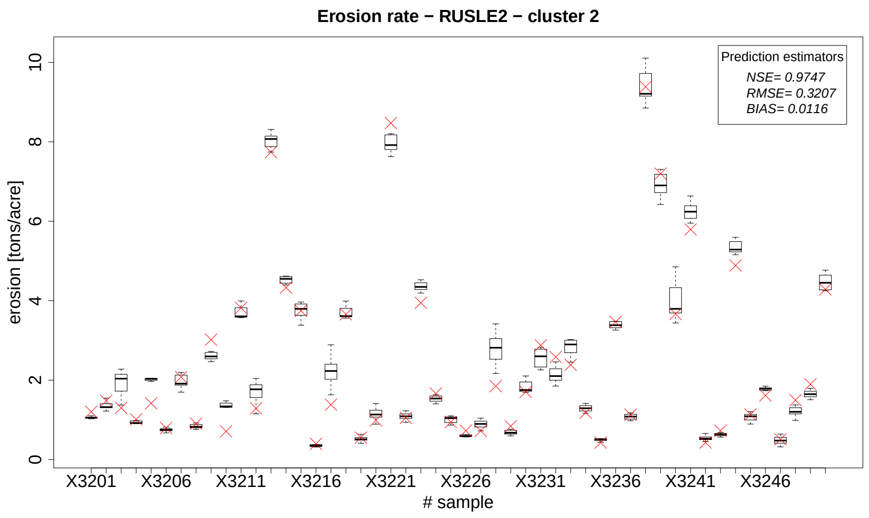


Figure 56: Cluster 2. Boxplots represents the ensemble of SMs estimates against RUSLE2 erosion runs (red crosses).

Cluster STIR $\geq$ 53 and erosion $>$ 11 tons/acre

Figure 58 and Figure 59 show performance of third eSM cluster. Figure 58 shows the actual comparison between observed (original model runs are represented with a red cross) and eSM emulated uncertainty quantified (boxplots represent eSM runs) values. eSM estimates are pretty accurate since NS efficiency is above 0.98, RMSE is 0.37 and BIAS =0.0683. The accuracy is mirrored in estimated vs original model erosion values laying on a 1:1 line in the scatterplot.

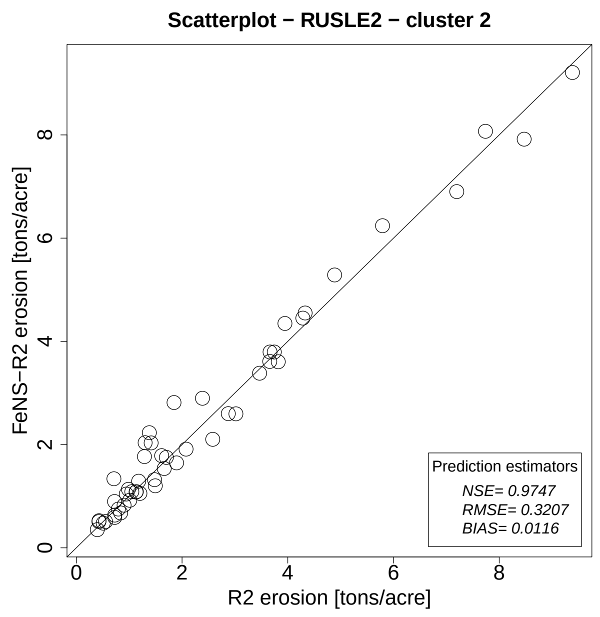


Figure 57: Cluster 2. Scatterplot of SM estimates (computed on the median of each boxplot) and RUSLE2 simulated values.

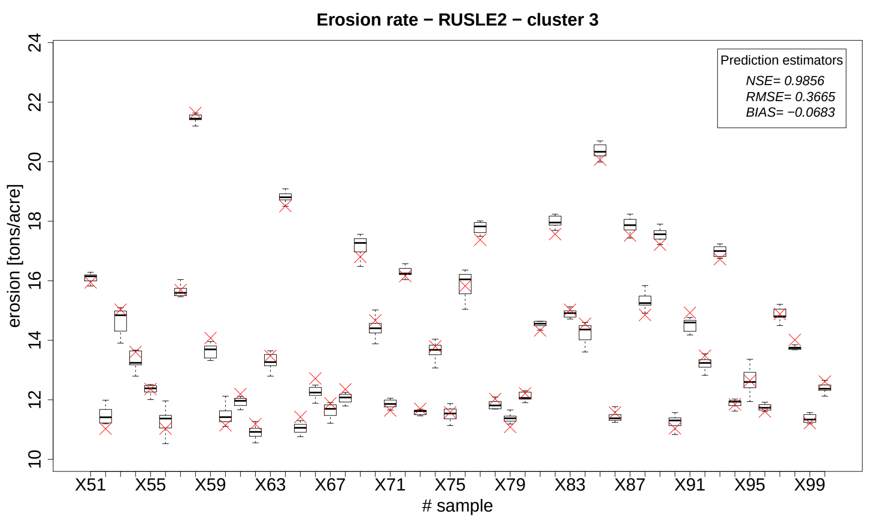


Figure 58: Cluster 3. Boxplots represents the ensemble of SMs estimates against RUSLE2 erosion runs (red crosses).

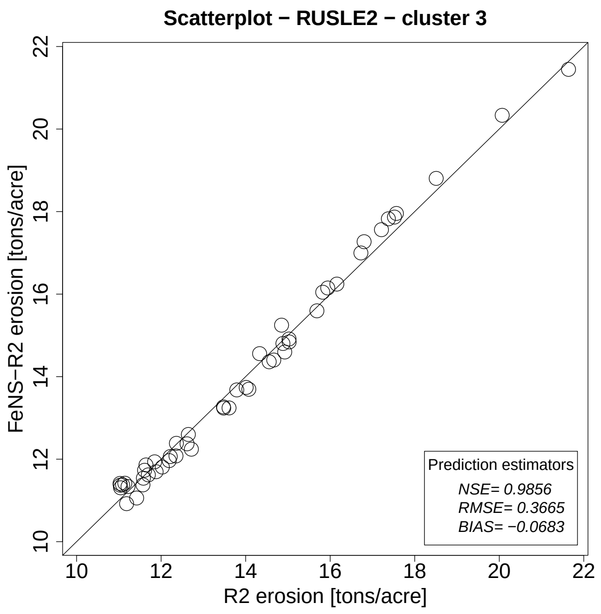


Figure 59: Cluster 3. Scatterplot of SM estimates (computed on the median of each boxplot) and RUSLE2 simulated values.

One single gate ANN was trained to forecast erosion rate higher or lower than 10.5 tons/acre based off of identical eSM input data structure. When run against the testing dataset, the gate ANN was wrong 14 times only over 9 754 samples.

3.4.1.3.4 DoE 3 - Step 4

DoE3 demonstrates the goodness of FeNS methodology even by enlarging input space dimensions by adding field length and crop yield. Despite reduced number of input parameters compared to original model input, FeNS is capable of emulating original model behaviour with respect to erosion rate.

Analyzing the identical input snapshot fed to eSM, the gate ANN has proved to be capable of accurately forecasting high or low erosion rate. However, one single gate ANN was trained instead of a ensemble gates and one single gating strategy has been developed. Clustering the input space is surely a useful technique. It allows for fastening eSM training by reducing TV dataset size, and improve estimate accuracy by splitting modeling behaviour responsibility between different cooperating eSMs.

Next DoE has to be designed on a larger study area and broader variety of soil properties, management and crops. Different gating strategies has to be developed to evaluate differences between options and pick the most accurate one. Clustering technique is surely important stage of data preprocessing but it is still a trial and error process without any sort of automation.

3.4.1.4 *Conclusions*

Despite reduced number of input parameters compared to original model entries, FeNS is capable of emulating original RUSLE2 behaviour with respect to erosion rate. Every DoE developed demonstrates goodness of FeNS methodology. These DoEs were milestones in supporting FeNS design, development, and testing.

However, DoEs can be improved by retrieving more homogeneous and better distributed model snapshots over the entire domain space and investigating into data clustering methodologies.

Data clustering is a useful technique since it allows for fastening eSM training by reducing TV dataset size, and improve estimate accuracy by splitting responsibility of modeling behaviour emulation between different cooperating eSMs. Consequently, the design of automated data clustering algorithm as part of FeNS pipeline is a fundamental improvement.

3.4.2 Agricultural Ecosystem Services (AgES)

AgES is a modular, Java-based, fully distributed watershed model (Ascough II et al. (2015b), Ascough II et al. (2015a), Green et al. (2014), Green et al. (2015)). It implements hydrologic/water quality modeling components for simulating daily water budget per hydrological response unit (HRU), in addition to plant, soil and nutrient processes interactions.

Even if it is less computational expensive that a full 3D physically based model (Green et al. (2014)), AgES still requires long computational time and a large input dataset to perform a simulation run. Five categories of input data are identified: spatial structure (described with topology, HRU attributes, channel reach and routing files); management (described with management, crop rotation, landuse, fertilizer, and till files); parameters (described with crop types, soil horizons properties, groundwater properties files); and climate data (precip, relative or absolute humidity, temperature, solar radiation, and wind speed files).

DoE for generating SMs of AgES are mainly focused on emulating daily runoff as a result of rainfall distribution in space and time, soil characteristic and topology of the modelled watershed, and groundwater storage capacity.

South Fork Iowa River SFIR is a watershed located in central Iowa, which covers an area of $581\,\mathrm{km}^2$ across Wright, Franklin, Hamilton, and Hardin counties (Green et al. (2018b)) (Figure 60). The average annual precipitation is about $850\,\mathrm{mm}$, while the average annual air temperature is $10.5\,°\mathrm{C}$.

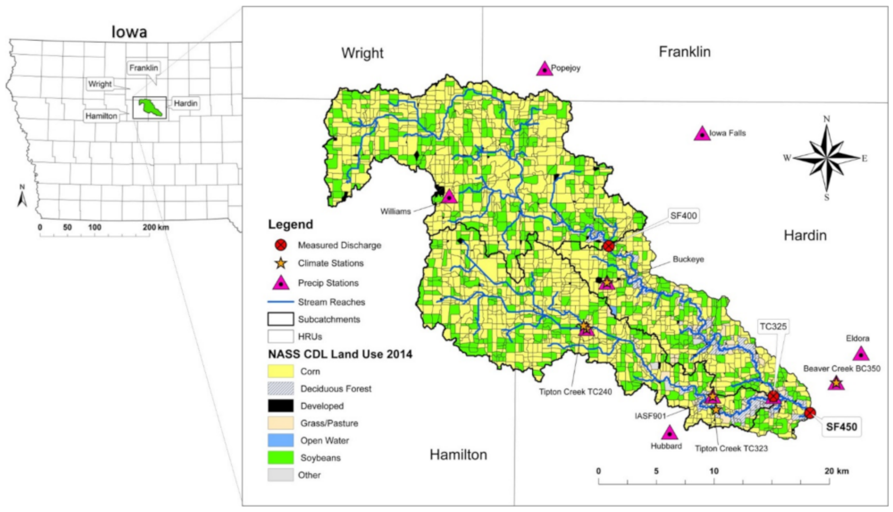


Figure 60: SFIR watershed across Wright, Franklin, Hamilton, and Hardin counties, credit Green et al. (2018b).

SFIR is of particular interest for United Stated Department of Agriculture (USDA) Conservation Effects Assessment Program (`https://www.ars.usda.gov/`

anrds/ceap/iowa-southfork/) since it is used as a benchmark watershed in the Corn Belt for monitoring environmental impact of intensive livestock management and usage of agro-chemicals to grow rainfed crops (Tomer et al. (2008)).

The Southfork Watershed Alliance (SWA) (http://www.southforkwatershed.org/) demonstrates the actual regional interest in preserving SFIR environmental integrity. The SWA is an advisory team that works in close contact with stakeholders to encourage agricultural management and conservation practices.

Consequently, service delivery organization such as USDA-NRCS and potentially SWA are interested in SMs capable of emulating AgES behaviour for SFIR water quantity/quality scenarios to support stakeholder decisions on the field.

This DoE is the first attempt to exercise FeNS methodology on a fully distributed watershed model. Consequently, further investigation on expanding FeNS-AgES capabilities on nitrogen prediction becomes of great importance to monitor stream water quality.

3.4.2.1 *DoE*

This experiment aimed to test the capability of FeNS system to recognize and emulate the hidden nonlinear function that describes AgES behaviour with respect to predicted daily runoff at the outlet of the SFIR watershed.

3.4.2.1.1 DoE - Step 1

AgES was properly calibrated to estimate streamflow and nitrogen quantity at catchment outlet as a result of cropland management practices (Green et al. (2018b)). The entire watershed was divided into 3 015 HRUs, and since SFIR watershed is characterized by photole depressions, 1 948 HRUs were modelled including tile drainage AgES module (74% of the SFIR watershed). 99% of the cultivated area produced corn and soybean. AgES simulations run daily from Saturday 1st January, 2000 until Thursday 1st January, 2015. Further details are thoroughly described in Green et al. (2018b).

3.4.2.1.2 DoE - Step 2

To emulate AgES behaviour with respect to current day runoff $Q(t)$, only the most sensitive parameters were selected. A total of 7 fully lumped input parameters were identified and following listed:

- Precipitation - $P(t)$;
- Leaf Area Index - $LAI(t)$;
- Potential Evapotranspiration - $PotET(t)$;
- Variation of snow depth - $\Delta SD(t, t-1)$;
- Groundwater level - $GW(t-1)$;
- Soil Saturation - $SSat(t-1)$;
- Runoff - $Q(t-1)$.

These parameters result from the arithmetic mean of actual model input values computed on the centroid of each HRU. The only exception is Runoff - $Q(t-1)$ which is estimated at the watershed outlet.

As a result, current watershed runoff was expressed as:

$$\begin{aligned} Q(t) = f(P(t), LAI(t), PotET(t), \Delta SD(t,t-1), \\ GW(t-1), SSat(t-1), Q(t-1)), \end{aligned} \tag{3.4}$$

where $P(t)$ is the current meteorological forcing; $LAI(t)$ and $PotET(t)$ characterize current state of plant/crop canopy; $\Delta SD(t,t-1)$ represent the variation of snow storage between current and previous day; $GW(t-1)$, $SSat(t-1)$, and $Q(t-1)$ identify the actual state of the watershed and ideally provide an estimate of stored water. Since the response time of the watershed is about 1 day, no information older than previous day have been taken into account. Figure 61 shows the final eSM setup.

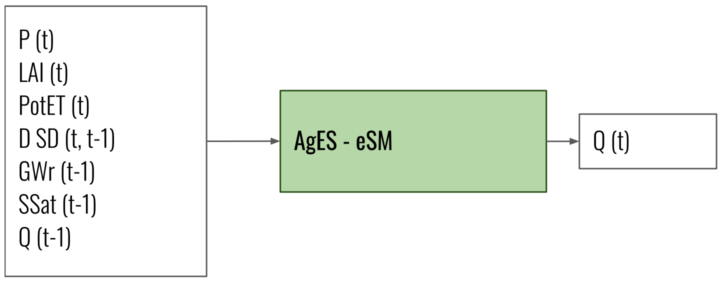


Figure 61: Generic SM input/output structure for DoE1.

The collected AgES snapshots were split into 90% TV dataset (Saturday 1st January, 2000 - Sunday 31st March, 2013) and 10% testing (Monday 1st April, 2013 – Monday 16th June, 2014). Each SM was generated using the following hyper parameters:

- Initial population = 5 000
- Connection density = 0.1
- Final training error = 10^{-4}
- Training dataset = 90%
- Validation dataset = 10%
- Splitting mechanism = "Same Distribution"

3.4.2.1.3 DoE - Step 3

Figure 62, Figure 64, and Figure 66 show FeNS - eSM performance for runoff $Q(t)$ estimates. These plot illustrate eSM emulation of four peak runoff discharges out of the entire testing dataset (the entire testing set is 442 timestamps). Prediction estimators are computed on the entire testing set, instead.

In Figure 62, Figure 64, and Figure 66, red crosses indicate original AgES runoff computation, while boxplots indicates FeNS-eSM uncertainty quantified results. Prediction performance of the eSM are evaluated on the median of each boxplot.

FeNS-eSM is capable of emulating AgES model behaviour on the testing dataset. NS is above 0.97, RMSE is slightly below 2.25, and finally the BIAS is slightly negative -0.0794.

The AgES-eSM was built with 16 SMs with totally different structures: the smallest SM has 4 hidden nodes and 27 links, while the structure of the biggest SM consists of 9 hidden nodes and 52 connections.

Despite spatially lumped reduced number of input entries compared to original model parameters, FeNS is overall capable of emulating original AgES behaviour. However, it sometimes shows inaccuracy in reproducing the rising (Figure 62) and decreasing limb of the hydrogram (Figure 66). Further DoEs will investigate techniques for improving estimate accuracy.

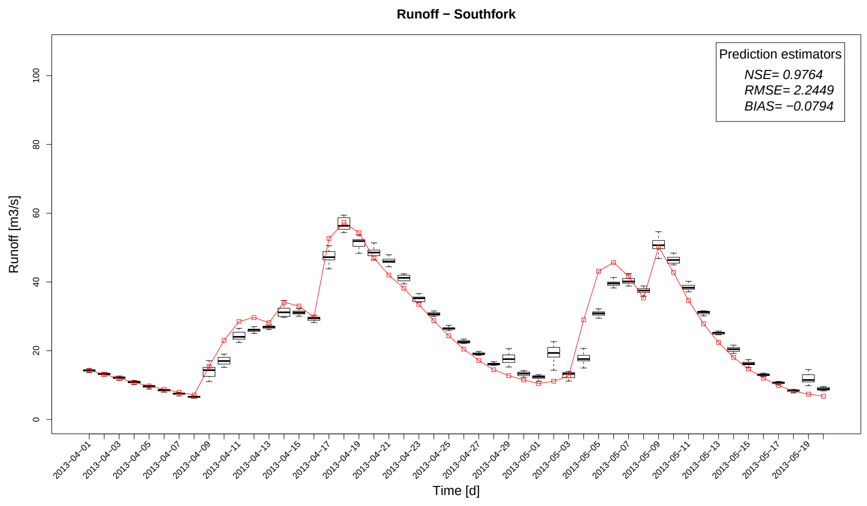


Figure 62: Boxplots represents the ensemble of SMs estimates against AgES runoff computations (red squares).

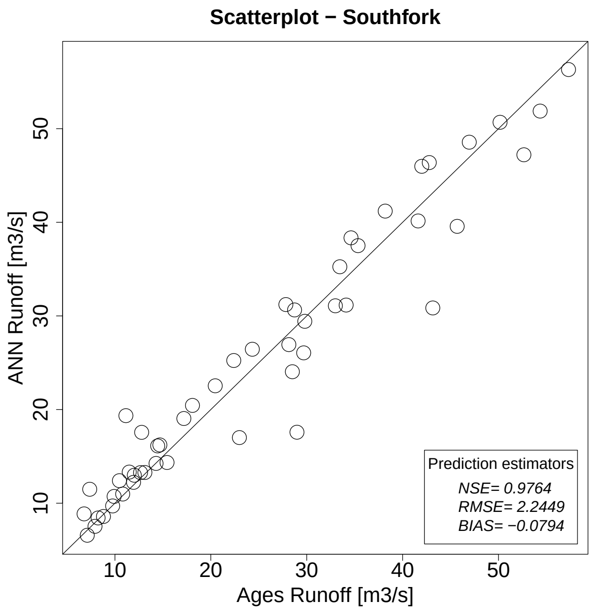


Figure 63: Scatterplot of SM estimates (computed on the median of each boxplot) and AgES simulated values.

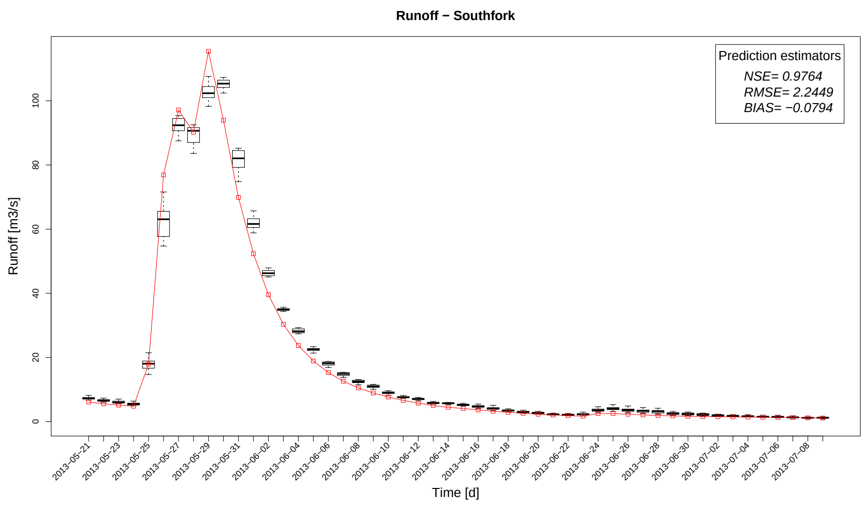


Figure 64: Boxplots represents the ensemble of SMs estimates against AgES runoff computations (red squares).

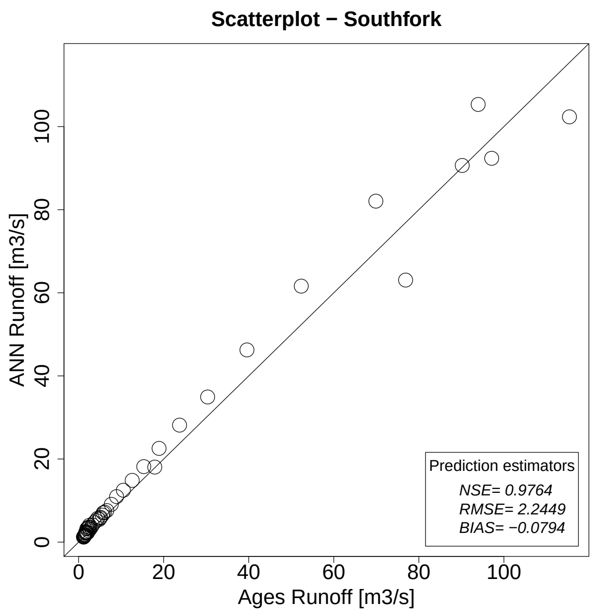


Figure 65: Scatterplot of SM estimates (computed on the median of each boxplot) and AgES simulated values.

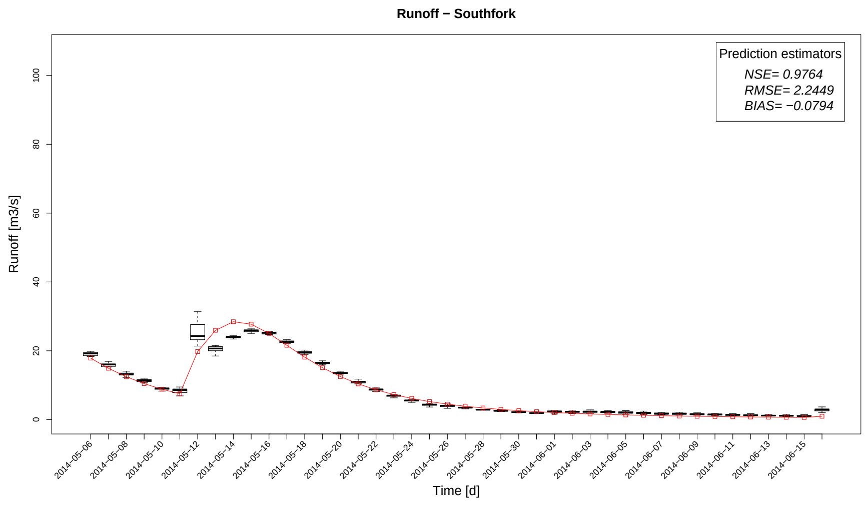


Figure 66: Boxplots represents the ensemble of SMs estimates against AgES runoff computations (red squares).

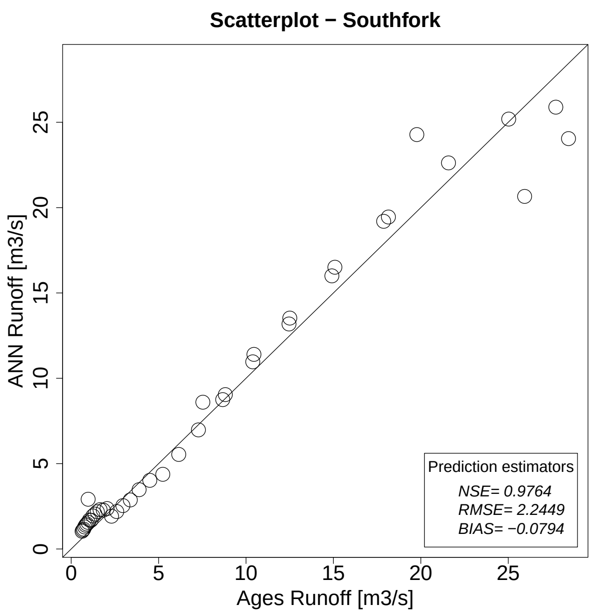


Figure 67: Scatterplot of SM estimates (computed on the median of each boxplot) and AgES simulated values.

3.4.2.2 *Conclusions*

Despite spatially lumped reduced number of input entries compared to original model parameters, FeNS is overall capable of emulating original AgES behaviour with respect to predicted daily runoff at the outlet of SFIR watershed.

However, with the purpose of supporting service delivery organizations such as USDA-NRCS and SWA on the field, capabilities of eSM for AgES model need to be expanded. Here, the eSM has to be trained to emulate AgES response with respect to nitrogen load, since monitoring the environmental impact of intensive livestock management and usage of agro-chemicals to grow rainfed crops is a big concern in the Corn Belt.

Additionally, since AgES computes runoff and nitrogen load per HRU, FeNS can potentially scale down and generate eSM for each HRU. This improves decision making process support by providing service delivery organizations with estimates of runoff and nitrogen load on a field scale.

3.5 SUMMARY

This chapter introduces to design and implementation of Framework enabled NEAT-based Surrogate modeling (FeNS) approach.

Literature review illustrates motivations behind the choice of building the FeNS system on top of ANN data-driven empirical surrogates, instead of leveraging projection based or multifidelity methods. Literature review also identifies drawbacks of ANN surrogate methodologies but underline previous research that values this approach for decision support and integrate modeling.

Afterwards, research questions identify the milestones that drive the development of FeNS.

Research design and methods describe methodological and technical approaches utilized to achieve the automated generation of SMs at a framework level. This section introduces to NeuroEvolution of Augmenting Topology (NEAT) and Feature Selective NEAT (FS-NEAT) and analyzes the genetic evolutionary algorithm that allows for automatically emerge SM from provided original model input/output snapshots. To take full advantage of the stochasticity involved in the evolutionary algorithm, a cross validation-like procedure is specifically designed and analyzed. This allows for emerging an ensemble of surrogate models and uncertainty quantifying eSM results. Finally, these methodologies are actually integrated at a framework level. FeNS concept allows for identifying the protocol that rules the merging of previously described methodologies into framework workflow without changing user approach to standard simulation model workflow. FeNS architectural design illustrates the software elements that facilitate the integration of evolutionary algorithm and ensemble of ANN in the modeling framework, and automatically emerge the eSM. The technical approach describes MongoDB features, microservice architecture and RESTful API, Cloud Service Integration Platform (CSIP) API, Encog ML library, and finally CSIP-services the FeNS system is build upon.

DoEs for emulating RUSLE2 and AgES demonstrate goodness of FeNS methodology and allows for identifying future developments and improvements.

Next chapter describes the integration of NET3 approach into OMS3 to facilitate research model maintenance, development and application. Literature review and research questions determine starting point of the study and path that guides through the research respectively. Research methodologies and case studies describe methods employed to carry out the integration of graph theory applied to a graph modeling structure into OMS3 and consequent application to actual test cases.

4 COMPLEX NETWORK BASED PHYSICAL MODELING

Contents

4.1 INTRODUCTION

The problem statement of this dissertation highlights constraints and limitations research scientists face while dealing with environmental simulation model code base. Here, the term "operational use" applies to model maintenance and development, and state-of-art consultancy applications.

The integration of last enhancements in terms of conceptual model design, numerical integrations, physical processes descriptions, GIS capabilities, and other tools to already existing code base is fundamental part of research efforts and advancements. This improves environmental models results accuracy in simulating natural phenomena.

However, further integration of innovative engineering design practices or physical processes descriptions become cumbersome and counterproductive due to increasing software code base complexity and lacking of proper software architectural design. Additionally, these aspects complicates the application of simulation models itself since they increase requirements of IT computational infrastructure management, data preparation proficiency, and difficulty in understanding code base implementation of modelled processes.

Summarizing, operational use of environmental simulation models is not a smooth and straightforward process due to inner complexity of model code base. As a consequence, further research enhancements get slowed down.

The adoption of EMFs facilitated both environmental simulation model maintenance and development, and their application. EMFs were originally designed and promoted to foster separation of software architectural design from scientific content. A framework enables a sort of software "plug-ins" system: each conceptual/physical process is encapsulated into a single stand alone software application and the framework manages interconnection of modeling applications as part of a whole modeling solution. The system plug-in design simplifies the creation of new modeling solution by allowing for easily swapping out a model component from the original modeling solution for a different one.

This software architectural concept helps transitioning modeler creativity from mathematical equations into component implementations and model creations.

By the time EMFs were released, modelling an entire watershed by swapping out components to tests different modeling approaches was a very innovative research methodology. However, this constraints scientists to model a watershed as an homogeneous entity. To account for watershed heterogeneity and allow for further modeler creativity, EMFs features need to be expanded.

OMS3 is state-of-art in terms of EMFs, and motivations behind this statement are subjects of further deep investigation in subsubsection 4.4.2.1, *Object Modeling System v3 (OMS3)*. To improve and expand OMS modeling flexibility, a directed acyclic graph (DAG) modeling structure (NET3) (Serafin et al. (2017, 2018a)) is implemented and fully integrated into the OMS core. This allows for connecting interrelated OMS modeling solutions and run them as a whole, more complex system. The initial prototype has been developed upon the semi-distributed hydrological system GEOframe. Thus, implementation and main features derive from modeler creativity needs in terms of hydrological modeling.

However, further examples demonstrate NET3 flexibility of modeling any complex network based applications.

JSWMM is a Java component based redesign of Storm Water Management Model (SWMM), which allows for both designing and verifying a storm sewer network. JSWMM architectural design takes advantage of the benefits of OMS3-NET3 capabilities of avoiding code duplication and implicitly parallelize independent mathematical computations. JSWMM has been developed as part of the Urban Hydrology module of the GEOframe environment.

System of Systems of Models (SSoM) is a software application developed for Framework for Integrating the Complexity of Uncertain Systems (FICUS) project. SSoM makes use of OMS3-NET3 capabilities of encapsulating completely different OMS modeling solutions in different nodes of the graph modeling structure, and interconnecting them as whole system. Furthermore, SSoM specifically leverages OMS multi-language interoperability of connecting Java, Python, and R OMS compliant components (Serafin et al. (2018c)).

4.1.1 River network – graph structure analogy

The requirement of improving EMFs modeling flexibility with graph modeling structure capabilities derives from river networks – graph structure modeling analogy (Figure 68).

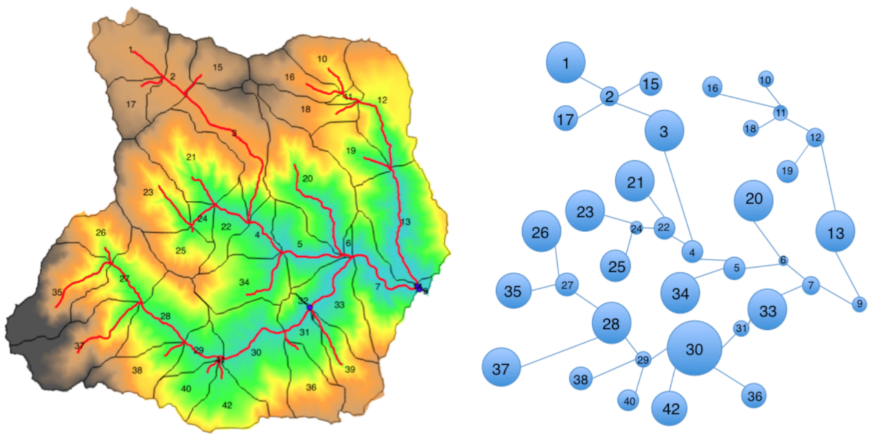


Figure 68: Representation of the river network – graph structure analogy, credit Bancheri (2017)

A river network subdivides a catchment in interconnected units. Here, each unit fosters a large variety of natural phenomena. From modeling perspective, each natural process can be conceptualized into a model component. Consequently, a modeling solution happens to connect model components to reproduce natural phenomena interactions at unit scale.

Finally, a graph modeling structure orchestrates modeling solution interconnections, which reproduces natural interaction of interrelated units following river network topology.

The landscape of a watershed landforms with valleys and a dendritic river network (Howard (1994)). Hillslopes are identified by convex to linear topography while the river network is identified by interconnected channels delimited by river banks (Montgomery and Dietrich (1989); Montgomery and Foufoula-Georgiou (1993); Howard (1994); Hooshyar et al. (2016)). Proper modeling of an heterogeneous drainage basin involves the analysis of size/scale of slopes and related valley networks (Ehlschlaeger (1989); Montgomery and Foufoula-Georgiou (1993); Demir and Szczepanek (2017)). Thus, accurate analysis of watershed hydrological behaviour begins by splitting the entire study area in homogeneous units.

Several studies have been carried out to investigate different strategies to identify channels head (Montgomery and Dietrich (1988); Montgomery and Foufoula-Georgiou (1993)), delineate the river network and subdivide an entire watershed in subwatersheds and interwatersheds starting from DEMs (Ehlschlaeger (1989); Montgomery and Foufoula-Georgiou (1993); Hooshyar et al. (2016); Demir and Szczepanek (2017)). Older algorithms require channel initiation thresholds (slope-area (Dietrich et al. (1993); Ijjasz-Vasquez and Bras (1995)), Strahler's order (Peckham (1995)), contributing area (Band (1986); Tarboton et al. (1991))). Innovative methods take advantage of the benefits recent availability of DEMs with resolution lower than 3 m (high resolution DEMs) to more accurately estimate geomorphologic and hydrologic features. Here, openness (Sofia et al. (2011)), slope direction (Lashermes et al. (2007)), curvature (Sofia et al. (2011); Pelletier (2013)), and curvature with k-means clustering (Hooshyar et al. (2016)) are the main topographic attributes used.

These hydro-geomorphic analysis can be automated and several Open Source GIS software applications have been developed, e.g. JGrasstools and the Spatial Toolbox in uDIG (Abera et al. (2014); Formetta et al. (2014a)), LandSerf (Wood (2009)), Geospatial Analysis Tools (Lindsay (2005)), TauDEM (Tarboton (1997)), GRASS GIS (Jasiewicz and Metz (2011)), and GeoNet (Passalacqua et al. (2010b,a)).

These methods and software applications allow for accurately characterizing river network and subwatersheds with minimal human impacts (Hooshyar et al. (2016)). Subwatersheds, or analogous entities such as hillslope or HRU, can be considered independent and interconnected by hydrodynamical networks (surface flow, interflow, and groundwater flow) (Grübsch and David (2001)).

From a modeling perspective, it is possible to bundle the modeling behaviour of a subcatchment and related channel into a single and independent entity (mass/energy storage compartments) which shares fluxes (mass, energy fluxes and exchanges) with connected entities (Phillips et al. (2015)). Consequently, the natural river network is conceptualized into a tree-like structure of interconnected entities.

However, human infrastructures have a relevant impact on current river network topologies. Since derivations, hydropower, and other artifacts are heavily disseminated over a watershed and its subwatersheds consequently, a tree-like structure doesn't provide enough modeling flexibility.

A graph-like structure enables a further layer of modeling adaptability to conceptualize natural river network as well as human infrastructures into a complex modeling solution.

Here, the mathematical standpoint hinges to graph theory, which allows for describing and analyzing any network and its properties (Heckmann et al. (2015)). Graph theory in environmental applications is an emerging field of research (Phillips et al. (2015)).

The actual programming implementation comes down to graph data structure, its traversing, and parallel computation of independent nodes.

When a graph data structure is implemented into a modeling framework such as OMS3, each node is enabled to handle modeling solutions instead of data only.

The next section reviews scientific literature with respect to applications of graph data structure to environmental modeling problems. The Research Questions section highlights the main points this dissertation tries to answer to. The Research design introduces to the methodologies used to pursue the objectives of this research: the methodological approach summarizes directed acyclic graph data structure, environmental modeling frameworks, and implicit parallelism from a theoretical standpoint; the technical approach deeps down to the motivations of expanding OMS3 computational capabilities and NET3 actual implementation.

4.2 LITERATURE REVIEW

The application of graph theory and related computational algorithms in hydrological modeling and engineering network problems is a long-studied concept.

One of the first applications dates back to Apostolopoulos and Georgakakos (1997).

Here, the authors design a parallel algorithm built upon a tree-like topology of the drainage network to speed up the computation of streamflow predictions with distributed hydrologic models. At that time, parallel computers and distributed computing environments were recent technological enhancements and parallel algorithms were innovative methodologies to satisfy computation needs of real-time forecasting of floods and flash floods events.

Following publications focused on improving parallel algorithms in order to achieve higher efficiency and computational performance.

For the sake of distributed modeling, Grübsch and David (2001) propose a heuristic *divide and conquer* algorithm to deal with computational challenge of subdividing water catchments into (theoretically) unlimited number of subareas. Here the goal is to make use of graph theory, and more precisely tree-like topology, to properly distribute the computational load among networked multiprocessors computer clusters. The authors develop a graph partitioning algorithm to group set of subareas and assign them to specific processors, consequently limiting interprocessor-communication overhead.

Vivoni et al. (2005) parallelizes a fully distributed hydrological model by leveraging graph-based domain decomposition of a water catchment into interconnected subbasins. He proposes a message passing interface (MPI)-based version of triangulated irregular network (TIN)-based Real-time Integrated Basin Simulator (tRIBS) and demonstrates performance and efficiency compared to a sequential version.

Afterwords, several publications propose MPI-based parallelization of hydrological models based on spatial domain decomposition of a watershed into interrelated subcatchements.

Wang et al. (2011) proposes the mapping of a drainage network into specifically designed binary-tree structure: when three or more upstream sub-basins flow into the same stream reach, a virtual node with no topological features works as joint of two upstream sub-basin and then merges with the remaining sub-basin. Here, the actual implementation is based on a MPI master-slave architecture with a centralized database that works as system data center.

Li et al. (2011) discusses another MPI master-slave computational approach based on dynamic decomposition of a drainage network to better control load balance.

Liu et al. (2016) proposes a two-level parallelization method for improving fully-distributed model computational scalability by leveraging MPI API.

However, every contribution is completely focused on improving parallel algorithms efficiency, computing time, and speedup ratio. There is no mention on modeling flexibility, code reusability, operating system interoperability, and the concept of separating software architectural aspects (e.g. implicit parallelization) from scientific contents is not a concern. Actually, Li et al. (2011) states that

> [...] since the parallel programs and simulation models are blended, neither the frameworks [MPI] nor the codes for parallelization can be reused by other models.

In conclusion, to take advantage of previously described parallel computational frameworks or algorithms, a research scientist needs to develop dedicated programming proficiency.

4.3 RESEARCH QUESTIONS

This section introduces to the research questions investigated by this dissertation. Each research question is briefly analyzed. Two research subquestions (RQ3.a and RQ3.b) are also proposed but not investigated and remain open questions for future research work.

RQ1: What is the best strategy to simplify development and run of conceptual/physical model?

Environmental modeling frameworks introduced three main revolutionary concepts in the scientific modeling community: (1) software encapsulation, (2) plug-in system, and (3) code duplication avoidance. The main goal of this dissertation is to elevate these concepts to a further layer of flexibility and abstraction.

The introduction to this chapter states that graph theory, and its actual implementation, can be applied to improve hydrological modeling at basin scale. Watersheds with minimal human impact are easily mapped to tree structures where nodes are mass/energy storage compartments and links are mass/energy fluxes. Various applications/frameworks in the past implemented this structure already (Grübsch et al. 2001, Li et al. 2010, Zaliapin et al. 2010, Cui et al. 2011, Wang et al. 2011, Demir et al. 2017).

However several contributions highlight how engineering works affect stream network and watershed modeling consequently (Gregory 2006, Whol 2006). Dam construction, diversion, culvert or draining systems, and mini-hydro power plants involve changes in flux directions and require dedicated modeling (Gregory 2006, Whol 2006). For example, modeling a run-of-river mini-hydro power plant necessitates of accurate design. Here, the modeler may want to simulate different scenarios to estimate the amount of:

1. electric power generated during the working hours of the turbines in the powerhouse while monitoring the impact of water diversion on physical and ecological conditions along the diverted reach;
2. sediment flowing into the de-silting box;
3. electric power generated based off of the amount of water diverted into the penstock, which depends itself on the hydrodynamic conditions of the downstream reach.

A tree data structure is not suited for modeling these processes because it doesn't allow a node (downstream reach) to have out-connection to more than one node (two in this specific example: run-of-river mini-hydro power plant and diverted stream reach, Figure 69).

Additionally, a tree-like structure limits modeler creativity when it comes to model HRUs instead of subwatersheds. An HRU might have several other HRUs flowing in and it might flow out to more than one HRU and stream reach.

This research identifies with a directed acyclic graph (DAG) data structure, the most suitable tool for elevating EMFs modeling flexibility. A DAG allows for:

1. potentially encapsulating a different modeling solution in each node of a whole modeling structure;
2. easily plug in and out nodes in the entire graph modeling structure;
3. avoiding code duplication by re-using the same packaged source code for identical modeling solutions used in different nodes of the graph structure;
4. managing n-inputs and n-outputs for each node.

RQ2: What is the most suitable EMF core to expand and make more flexible?

Object Modeling System v3 (OMS3) David et al. (2013) has been identified as state-of-art in terms of environmental modeling frameworks.

OMS3 design is based off of the following notable software engineering approaches David et al. (2013):

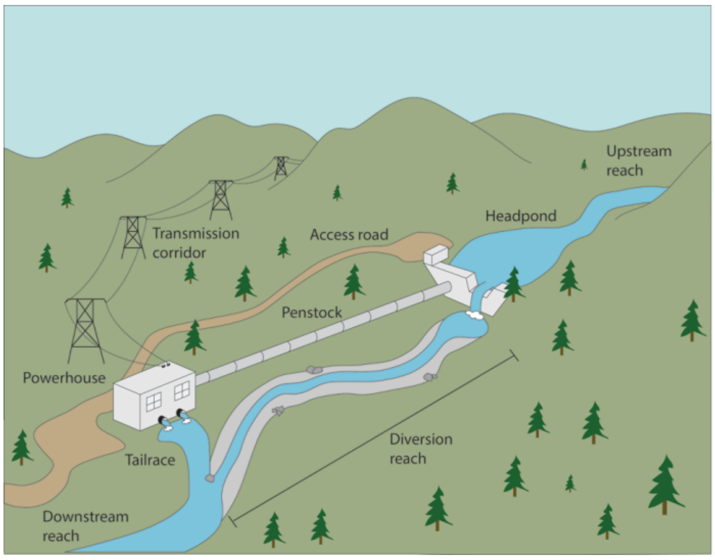


Figure 69: Generic schematization of a hydropower plant. Part of the stored water flows through the penstock from the dam, while the remaining flow through the diversion reach. Credit, Palen Lab blog.

- non-invasive lightweight framework design (modelers don't have to import and learn complex APIs);
- component-based model development;
- graceful adaptation/integration of legacy models or already existing code base;
- automated generation of model documentation by leveraging component metadata;
- implicit parallel computation of independent model components;
- modeling solution setup through user-friendly, flexible, groovy-based Domain Specific Language (DSL);
- integration with cloud-based platform CSIP (Lloyd et al. (2012); David et al. (2014a)) to enable computational scalability.

The design of OMS3 has been driven by the following concept:

> Environmental model development needs to be creative, i.e., new approaches have to be explored that go beyond the boundaries of given programming languages, data structures, algorithms, and existing architectures. EMFs should foster creativity, and not constrain the modeler to the framework developer's view.
>
> – David et al. (2013) –

Several scientific publications demonstrate the effectiveness of modeling with OMS-compliant components (Abera et al. (2017a,b); Ascough II et al. (2012); Bancheri (2017); Bancheri et al. (2018a,b); Dalla Torre et al. (2018); Formetta (2013); Formetta et al. (2013b,a, 2014a,b, 2016a,c); Green et al. (2015)). As a result, OMS3 is the most suitable EMF for hosting the development of NET3.

RQ3: What is the most effective strategy to fasten computational time of complex simulation models without requiring researchers to develop any specific parallel software development skill?

The parallelization of several hydrological models is implemented upon a master-slave computing approach with MPI APIs (Hluchy et al. (2001); Rao (2005); Cui et al. (2011); Wang et al. (2011)). Fine development of MPI-compliant algorithms requires programming proficiencies and deep knowledge on shared memory and computational workload management. Exclusively accurate design allows for properly scaling across computer clusters and exploiting their entire computational power (Tran and Hluchy (2004)). Furthermore, MPI-based software application has to compile on each hosting operating system and hardware, constraining software portability.

More recent applications implement against Graphics Processing Unit (GPU), innovative hardware solutions that allow for highly parallel computation (Kalyanapu et al. (2011)). Once again, running GPU-compliant software requires NVIDIA™ graphic cards, limiting software portability.

In terms of software implementation, developing against MPI or CUDA® APIs necessitates of dedicated programming knowledge and hardware.

OMS3 is a Java-based framework. As a result, any kind of portability issue is resolved since the Java Virtual Machine (JVM) runs on top of any operating system. Java parallelization entry level is multi-threading, which allows for parallelizing software applications on one single computer by leveraging multi-core Central Processing Unit (CPU)s (David et al. (2013)). Parallel computation in hydrological models is hardly achievable because of their complexity. OMS3 enables implicit parallelism of independent components in a modeling solution. OMS3 manages creation and intercommunication of computational threads: independent components are executed by different threads while threads communicate through data flow between connected components. Encapsulating each conceptual/physical process of the hydrologic cycle into a single component is the only model developer's responsibility. Then, the model user describes component connections through the modeling simulation file. Eventually, OMS3 automatically parallelizes the run of independent processes (or OMS-compliant components), thus requiring no parallel programming skills to model developers.

OMS3 takes advantage of the inherent indepence of conceptual/physical processes of a modeling solution to implicitly parallelize simulation run. NET3 enables a further layer of implicit parallelization by leveraging the graph structure topology to concurrently run modeling solutions of independent nodes.

RQ3.a: Can this further layer of implicit parallelism effectively speed up the computation of both small and large scale modeling solutions?

This dissertation won't be able to investigate this research question. It remains an interesting open question which requires dedicated work to properly demonstrate if this further layer of implicit parallelism speeds up the computation of smaller and larger complex modeling solutions.

RQ3.b: What is the proper trade off between graph topology and component connections related parallelizations?

This dissertation won't be able to investigate this research question. It remains an interesting open question which requires purposeful work to properly define the most effective trade off between the number of computational threads to dedicate to component-connections parallelism and the number of computational threads to dedicate to graph topology parallelism.

4.4 RESEARCH DESIGN AND METHODS

This research aims to accommodate modelers and researchers requirements of facilitating environmental model development and use. To achieve this goal, EMFs capabilities are extended by implementing a fully integrated Graph Modeling Structure (GMS).

The GMS originates from a standard DAG where each node runs a modeling solution. Afterwords, the GMS connects modeling solution results into a complex topology-driven modeling solution.

Thus, this research has been driven by the need of:

- Simplifying
  - the development of complex modeling solutions;
  - the runs of complex modeling solutions;
- Improving the flexibility of actual EMFs to accommodate modelers creativity and talent;
- Reducing computational time by enabling a further layer of implicit parallelization.

Subsection *Methodological approach* introduces to three main methodologies this research is built upon:

1. **Directed Acyclic Graph data structure (DAG)** is the data structure that manages interconnected OMS3 simulation files in topological order;
2. **Environmental Modeling Framework** are modeling software that allow for completely decoupling model code development from framework architectural infrastructure and design;
3. **Implicit parallelism** is a framework functionality that tacitly parallelizes the computation of independent modeling components.

Subsection *Technical approach and implementation* comprehensively introduce to the architectural design of the Object Modeling System v3 (OMS3) environmental modeling framework. This preamble allows for deeply describing the implementation of the Graph Modeling Structure (GMS) NET3.

4.4.1 Methodological approach

4.4.1.1 *Directed Acyclic Graph data structure (DAG)*

This section introduces to basic notions of graph theory from discrete math and basic requirements to implement a directed acyclic graph data structure from computer science related field.

The paragraph *Graph theory review* highlights reviews basic (e.g. node, edge, and path) and formal definitions (e.g. directed graph and undirected graph). The main sources are McCreary and Reed (1993), Grübsch and David (2001), and Cui et al. (2011).

In paragraph *DiGraph API and traversing algorithms*, discrete math definitions are translated into software design concepts. Here, the scope of this dissertation is narrowed down to directed graph since directed hydrological fluxes connect mass/energy storage compartments. Thus, brief introduction on digraph standard API, its representations and implementations are summarized and following utilized in NET3 technical approach. Concepts derives from chapter 4 of Sedgewick and Wayne (2011), and Cui et al. (2011).

4.4.1.1.1 Graph theory review

A graph G is a mathematical structure that consists of individual objects and links between those objects. It allows for modeling the pairwise connections between those objects (Sedgewick and Wayne (2011)).

Formally, a graph is an ordered pair $G = (N, E)$ where a finite set of nodes $N(G)$ (or vertices) are connected through a finite set of edges (or arcs) $E \subseteq NxN$. Every edge $e \in E(G)$ defines a relation between a unique pair of nodes $u, v \in N(G)$. A *null graph* is the smallest graph possible which has one single node with no edges.

A *path of length* n from v_0 to v_n in a graph G is a finite sequence of connected nodes $v_0, v_1, \ldots, v_n$ where $(v_i, v_{i+1}) \in E(G), i \in \aleph \mid 0 \leq i \leq n-1$. The number of edges between v_0 and v_n defines the *length* of the path. v_0 is the *start node* of the path while v_n is the *end node*. The set of every path in G is P_G. A *cycle* in G is a path $p = (v_0, v_1, \ldots, v_n)$ where $v_i = v_k \mid 0 \leq i < k \leq n$ that connects a node to itself. A *simple path* in G connects two vertices without revisiting any vertices or edges. A *simple cycle* in G connects a node to itself without revisiting any nodes or edges except for the starting or ending node. A graph is *acyclic* if P_G doesn't contain any cycle.

A graph G is *connected* if and only if $\exists p \in P_G \mid p = (u, \ldots, v) \forall u, v \in N$, thus the set of path P_G contains a path for any pair of different nodes u, v.

A *directed graph* (or *digraph*) is a particular class of graphs where the edge set $E(G)$ is ordered and every edge $e \in E(G)$ defines a binary relation between two nodes $(u, v) \mid u, v \in N(G)$. Node u is called *parent*, *tail* or *immediate predecessor* of v, while v is called *child*, *head* or *immediate successor* of u. The set of nodes S where each $s \in S$ has no immediate predecessors is called *source* of G.

$$S = s \in G \mid (v, s) \notin E(G) \ \forall v \in N(G). \tag{4.1}$$

The set of nodes M where each $m \in M$ has no immediate successor is called *sink* of G.

$$M = m \in G \mid (m, v) \notin E(G) \ \forall v \in N(G). \tag{4.2}$$

In a digraph, a node v is *reachable* from a node u if and only if $\exists p \in P_G \mid p = (u, \ldots, v)$. In a digraph, it is possible to the define the *outdegree* of a node u as the number of out-edges, that is the number of edges pointing from the node u. Conversely, the *indegree* of a node u quantifies the number of edges pointing to the node u, namely in-edges.

A *directed acyclic graph* (DAG) G does not contain any cycle in the set P_G.

A *tree* is a DAG where there is only one *sink* node m (called also *root*) and it exists a path $p = (u, \ldots, m)$ that connects every node $u \neq m$ to the root. Sources of a tree are called *leafs*.

An *undirected* graph is characterized by an edge set E of unordered pair of nodes: each edge $e \in E$ links two distinct nodes $u, v \in N$ in both directions. $u \neq v$ because self-loops are forbidden. Formally, if $G = (N, E)$ is a directed graph, an undirected graph G' is created from G by extending E with its inverse relation E^{-1}: $G' = (N, E \cup E^{-1})$. This allows for considering each edge in E and its backward edge.

4.4.1.1.2 DiGraph API and traversing algorithms

The API of a *directed graph* (DiGraph) is built on top of the data structure implemented. A DiGraph has three different possible representations:

- **Adjacency matrix:** it is a NxN array of binary bits. The entry x_{ij} is 1 if there is an edge between node i and j, 0 otherwise. The memory required to store this data structure is $O(N^2)$. The time required to access, add or remove an edge is $O(1)$. However, the entire matrix has to be reallocated and copied over if a node is added or removed. This operation is a $O(N^2)$ algorithm. This data structure is very useful when it comes to storing dense graphs.
- **Array of edges:** it is an array of length E. Each element of the array describes the edge by storing the ordered pair of vertices, starting and ending points of the edge. Here the data structure is stored in $O(E)$ memory, which linearly grows with the number of edges of the graph. Accessing or removing an edge is an $O(E)$ algorithm: edge IDs are not stored, thus the entire arrays is parsed to select the proper pair of vertices connected by the edge involved. A new edge is added with $O(1)$ effort.
- **Adjacency list:** it is an array of length N. Each element of the array contains one vertex and the sequence of its outedges. The memory usage is $O(N+E)$, which can get to $O(N^2)$ in case of a dense or fully connected graph. Thus, this data structure is better suited for sparse graphs rather than dense ones where an adjacency matrix is more performant. Access, insertion or deletion of a vertex require $O(1)$ to $O(N)$ effort, depending upon the data structure used to store the array.

The fundamental methods exposed by a DiGraph API are briefly explained in table 2.

Table 2: Standard API exposed by a generic public class DiGraph.

Returned DT	Method signature	Description
	`DiGraph()`	Default constructor
	`DiGraph(Topology topology)`	Reads in and builds the DiGraph from topology file
`int`	`getN()`	number of vertices
`int`	`getE()`	number of edges
`void`	`addEdge(`**`int`**` startNode, `**`int`**` endNode)`	add edge from startNode to endNode
`void`	`deleteEdge(`**`int`**` startNode, `**`int`**` endNode)`	delete edge from startN-ode to endNode
`boolean`	`edgeExists(`**`int`**` startNode, `**`int`**` endNode)`	check if an edge exists between startNode and endNode
`void`	`addNode(`**`int`**` newNode)`	add newNode to the Di-Graph
`void`	`deleteNode(`**`int`**` node)`	delete node from the Di-Graph
`boolean`	`nodeExists(`**`int`**` node)`	check if node is part of the DiGraph
`Iterable<Integer>`	`getChildren(`**`int`**` node)`	children nodes of node

(... continue to next page)

Returned DT	Method signature	Description
`DiGraph`	`reverse()`	returns a reversed DiGraph (edges are reversed)
`String`	`toString()`	@override

There are many more methods that can potentially be implemented to extend DiGraph API capabilities. However, they are mainly application specific and proper algorithms need to be accurately designed.

Next step is definition and implementation of searching algorithms and respective APIs (table 3).

Table 3: Standard API exposed by a generic public class SearchAlgo.

Returned DT	Method signature	Description
`void`	`compute(String direction, Integer source, DiGraph graph)`	Search for subbranches in the graph
`Boolean`	`hasPathTo(Integer vertex)`	Search if vertex is connected to source
`Iterator<Integer>`	`pathTo(Integer vertex)`	Return the path from vertex to source

Although properties of nodes and edges are easily accessed through the DiGraph API, properties related to the overall DiGraph result from the analysis of each and every vertex and related connections. Consequently, traversing the structure of edges and nodes becomes key in application-specific problems.

Searching algorithms usually have dedicated APIs as a consequence of decoupling data representation from processing components. These APIs are designed on composition (HAS-A relationship) to reference to an object by using instance variables. In other words, a fully built DiGraph is passed as argument to the searching algorithm constructor. Then, the algorithm queries the DiGraph to systematically examine nodes and edges properties, and move along node to node connections. The two most important algorithms are *depth-first* and *breadth-first* search.

The depth-first search (DFS) solves the source reachability (or connectivity) problem:

- **single-source reachability:** it checks if a path between a source node s and a target node u exists in the given DiGraph;
- **multiple-source reachability:** it checks if a path between a source node s in a set of source nodes S and a target node u exists in the given DiGraph.

Furthermore, it solves the *single-source paths* problem by identifying the path that connects a source node s and a target node u. Paths discovered by DFS depends on the data structure used to store the DiGraph and the type of recursive search algorithm implemented.

The breadth-first search (BFS) solves the *single-source shortest paths* problem by checking if a path between a source node s and a target node u exists and identifying the shortest one.

The difference between DFS and BFS algorithms sits in rule used to extract the next node to process from the storing data structure: DFS retrieves the most recently added node by leveraging a stack type data structure, while BFS retrieves the least recently added node by leveraging a queue type data structure. As a result, BFS deeply analyzes all the connections closer to the source node first before moving to further connections. Opposingly, DFS looks at the furthest connection from the source node first, moving to a layer closer connection only when finds dead ends.

DiGraph are widely applied to scheduling problems where a set of tasks are to be completed under *precedence constraints*: certain tasks must await previous tasks to complete before starting their run. Here, tasks are nodes of the DiGraph and directed edges schedule precedence constraints (or topological order).

In precedence-constrained type of problems, a DiGraph can't contain a directed cycle because it would end up in a infinite loop with no feasible solution. Consequently, the DiGraph becomes a Directed Acyclic Graph (DAG).

4.4.1.2 *Environmental Modeling Framework*

This section introduces to definition, architectural aspects and design of generic frameworks. It following focuses on EMFs and their additional design concepts.

A framework is a software library that simplifies the development of domain specific applications by providing reusable design (Gamma (1995); Lloyd et al. (2011)). It implicitly defines a set of rules for building each domain specific application, which results in applications with similar code structure and classes/objects partitioning (Gamma (1995)). It differs from a software library because it controls the overall program (set of applications) execution flow by applying the inversion of control design pattern.

A software framework elevates the concept of separation of concerns (SoC) by dealing with several complicated software architectural aspects like high performance computing and thread control, infrastructure constraints, programming language specifications, hosting environment constraints (operating systems and underlying platforms), etc. (David et al. (2013)). It abstracts these aspects to a level that is appealing for non-specialists. As a result, application development is streamlined and application designers can specifically focus on application domain functionalities (Gamma (1995)).

Development goal of software frameworks is design reuse over code reuse. However, the addition of accurate design of framework compliant components avoid code duplication and consequent drawbacks.

Software frameworks are being developed for supporting code development in different fields like financial modeling (Birrer and Eggenschwiler (1993)), decision support systems (Gachet (2003)), compilers for programming languages on specific hardware (Johnson et al. (1992)), graphical editors for music composition or mechanical CAD (Vlissides and Linton (1990); Johnson (1992)), and environmental modeling (Bernholdt et al. (2003); Hill et al. (2004); Blind and Gregersen (2005); Collins et al. (2005); Gregersen et al. (2007); Moore et al. (2007); Peckham et al. (2013); David et al. (2013)).

This dissertation focuses on EMFs, which are designed to facilitate environmental research scientists in developing and maintaining mathematical models.

The development of a conceptual/physical model requires the understanding of physical description of natural phenomena and software development skills. EMFs facilitate separation of these concerns by providing the research scientist with tailored libraries that abstract software architecture design from model implementation. In addition to previously listed framework generic capabilities, EMFs features include seamless access to data, encapsulation of conceptual/physical processes into functional units (o model components), management of components interconnection and intercommunication, conversion of physical units between connected components,

handling of temporal and spatial stepping, simulation data analysis by visualizing plot results (David et al. (2013)). Furthermore, EMFs provide tools for managing interoperability between different programming languages (Serafin et al. (2018c); David et al. (2013); Dahlgren et al. (2004)). This smooths the learning curve to approach an EMF by allowing research scientist to use their favorite language.
The adoption of EMFs as a standard practice has several advantages like:

- reducing model development cost and time by facilitating the integration of legacy models with new models and maintaining existing modeling practices;
- elevating model component reuse in different modeling solutions by avoiding code duplication and consequent error prone software maintenance and debugging;
- introduction of combination of QA/QC into model component lifecycle to prevent model bugs, errors or defects and improve software runtime quality and error traceability;
- repurposing model solutions for new business needs;
- promoting the concept of reproducible research by providing consistent and verifiable model results (Bancheri et al. (2018b)).

As pointed out in Rizzoli et al. (2008), the modeler should experience an immediate return on investment by adopting a framework designed to increase modeling productivity.
Eventually, EMFs elevate modeler creativity as well by leveraging the plug-in system of model components to facilitate the creation of different modeling solutions scenarios (David et al. (2013); Peckham et al. (2013)).

In conclusion, software frameworks facilitate domain specific application design by elevating the concept of separation of concerns: the management of software architectural design aspects such as high performance computing and hosting environment constraints are delegated to the modeling framework, while application developers focus on domain specific application design.

4.4.1.3 *Implicit parallelism*

This section briefly introduces to the concept of explicit and implicit parallelism. Formal definitions are provided since are used in section *Technical approach and implementation*.

State-of-art in terms of CPU is multi-core processor, which means that two or more autonomous processor cores (or processing units) are placed on a single chip package (Ovatman et al. (2011)).
These types of CPUs are installed on a large variety of devices starting from personal computers and smartphones. As a result, parallel programming in application software is a fundamental methodology that allows for speeding up the computational effort while fully taking advantage of the underlying hardware (Ovatman et al. (2011)).
The design of software application that integrates parallelized algorithms and management of thread execution is called explicit parallelism. This requires programming proficiency since the software developer masters the concurrent execution of parallel tasks, their synchronization and communication, and related memory management.

Explicit parallelism is not standard practice when it comes to design framework-compliant components. The framework actually manages these software architectural aspects. Consequently, research scientists or components developers are responsible for component implementation and can plan for proper software system decomposition and granularity (David et al. (2013); Peckham et al. (2013)).

This contrapposes to the previously mentioned methodology and it is called implicit parallelism. Here the framework identifies each component as a potential parallel task and schedules its execution based off of component interconnection. This methodology might not result in a optimal parallel efficiency since component developer has no control on the overall program execution, not even the single task consequently. However, programmer doesn't have to worry about processes communication and management and can improve parallelization effectiveness by finely tuning system granularity.

In summary, implicit parallelism is a software architectural aspect most modeling frameworks implements to allow model developer for taking advantage of state-of-art computational processing unit without requiring parallel programming proficiency. Implicit parallelism is a notable feature of OMS3 and is furtherly extended by NET3 to provide for a further layer of computational speed up.

4.4.2 Technical approach and implementation

The Technical approach and implementation section describes the technical methodologies that are part of this dissertation.

In the first sub section, the Object Modeling System v3 (David et al. (2013)) is introduced. OMS3 is an open source framework released under MIT licence. OMS3 has been designed and developed by Dr Olaf David at Colorado State University (Fort Collins, CO). The development of this EMF has been supported by USDA-NRCS and USDA-ARS. The Object Modeling System v3 (OMS3) sub section introduces to the software engineering design and notable functionalities that make OMS3 state-of-art in terms of EMFs. The author of this dissertation DOESN'T take any credit with respect to OMS3 framework code base in general, OMS3 architectural design, and OMS3 implementation. Dr David is the sole author of OMS3 (David et al. (2013)).

The second subsection "Graph Modeling Structure: NET3" introduces to the actual contribution of this dissertation. NET3 has been developed to expand OMS3 modeling capabilities. NET3 is released as part of OMS3 modeling framework. Consequently, NET3 is Open Source project released under MIT licence.

4.4.2.1 *Object Modeling System v3 (OMS3)*

The Object Modeling System v3 is a Java based integrated environmental modeling framework, which supports a workflow to develop and deliver environmental models to user organizations.

OMS3 allows for consistently and efficiently building science components, which are fundamental functional units to disaggregate a complex environmental model in (Lloyd et al. (2011); David et al. (2013)). It also supports modules calibration and testing for facilitating model component development, modification or adjustment as science advances, and repurposing for emerging customer requirements.

This section focuses on describing OMS3 unique features and providing an overall workflow example by stepping through code base details. The paragraph *Overview* introduces to the foundations of the OMS3 architecture and its most important architectural design aspects. The paragraph *Framework invasiveness* describes the use of Java annotations to enable OMS3 architectural design aspects and make OMS3 a lightweight framework. Furthermore, this subsection illustrates the model developer standpoint of leveraging OMS3 annotations to transform a Plain Old Java Object (POJO) into an OMS-compliant component. The paragraph *Simulation DSL* describes the actual interface between modeler and framework workflow: the Groovy-based DSL flexibility is demonstrated by analyzing the DSL functionalities and their mapping into OMS3 internal code base. The paragraph *Modeling concept (model) - `oms3.dsl.Model.java`* describes the core part of the OMS3 modeling solution. DSL concepts are illustrated as well as their mapping into OMS3 internal classe. Finally, paragraph *Simulation run* provides an overall workflow of the OMS3 internals for a modeling solution run.

4.4.2.1.1 Overview

OMS3 is a lightweight integrated environmental modeling framework. It supports the modeling process by streamlining model code development, providing for seamless model access to data, and data analysis and visualization.

OMS3 results from a complete redesign of OMS2. It minimizes model invasiveness, improves portability, adaptability and infrastructure integration (David et al. (2013)). Additionally, it simplifies component integration, emphasizes implicit auto-scaling of simulation models in multi-core and multi-processor environments, provides for modeling simulation traceability and integrity, and is capable of generating auto-documentation of models and simulations (David et al. (2013)). The framework addresses agencies traceability requirements with program tracking and financial management responsibilities. Agencies running simulation models can utilize the benefits of OMS3 to create auditable simulation trails based on Secure Hash Algorithms, which is a Federal Information Processing (FIP) Standard. A further relevant feature is the capability to auto-document a model and simulation structure into an open document standard such as Docbook5+.

Figure 70 illustrates the four foundations of the OMS3 principle architecture: (1) modeling resources, (2) system knowledge base, (3) development tools, and (4) modeling/simulation products. An overall introduction to the framework workflow is following summarized.

Modeling resources are databases, services, version control systems, or other repositories (David et al. (2013)). Knowledge base and development tools are part of the OMS3 core. The OMS3 system derives information out of the connected resources and transforms it into framework knowledge bases. The OMS3 development tools use this generated knowledge bases to create modeling and simulation products (David et al. (2013)). Modeling and simulation products are model applications (science components), simulations supporting calibration and optimization procedures as well as parameter sensitivity analysis, results visualization and statistical analysis, audit trails for reproducible research and legal purposes, and documentation.

The main architectural design aspects of the OMS3 framework are:

1. *runtime introspection* for parsing class structure, fields, methods, and their values, which allows the framework to hook into component entry points;

2. *annotation* of relevant class information, methods, and fields;

3. *reflection* as a methodology for accessing object fields and invoking object methods.

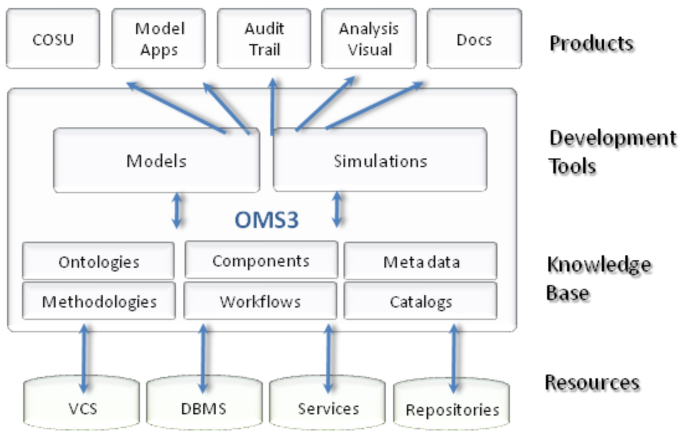


Figure 70: Schematization of OMS3 architectural design, credit David et al. (2013).

4.4.2.1.2 Framework invasiveness

This section introduces to generic framework invasiveness concept. Definitions of tight and loose coupling architectural design as well as heavyweight and lightweight framework are provided. Knowledge of these notions allow for understanding and appreciating OMS3 characteristics that make it a lightweight non invasive framework. An example of POJO annotation is provided. Additionally, Appendix B illustrates the adaptation of R and Python scripts into OMS-compliant components (Serafin et al. (2018c)).

Framework invasiveness is code coupling aspect that quantifies the level of dependencies a framework imposes to a compliant component (Lloyd et al. (2011)). Framework invasiveness correlates inversely with quality of modeling code (Lloyd et al. (2011)).

The definition of framework invasiveness resembles object-oriented coupling, which is the degree of dependencies between two software components (Gamma (1995)). Two classes are tightly coupled when they strongly depend on each other, and are hardly reusable in isolation consequently (Gamma (1995)). Opposingly, loosely coupled design allows classes to properly interact without deep knowledge of each other implementation (Freeman et al. (2004)). This design principle facilitates independent reusability and portability of two classes while elevating modifiability, extensibility, and overall maintainability (Gamma (1995)). Previous research on the topic showed an inverse correlation between object-oriented coupling and software fault proneness (Briand et al. (2000, 1999)).

Richardson (2006) provides definitions for heavyweight and lightweight frameworks, which essentially differ in the size of exposed API.

A heavyweight framework API has normally a considerable size. Consequently, familiarizing with this API requires time and expertise. It additionally generates extensive dependency with the application code.

Differently, a lightweight framework replaces massive APIs with alternative methodologies. This approach restrains the use of framework dedicated data types, interfaces, classes and it bounds the amount of boilerplate code in domain specific components.

Lloyd et al. (2011) thoroughly summarized the comparison between heavyweight and lightweight framework design, which resulted in table 4.

Table 4: Comparison between heavyweight (traditional) and lightweight frameworks, credit Lloyd et al. (2011).

Traditional framework	Lightweight framework
Components under the framework are: • bound statically at compile time • ightly coupled to the framework by extension of framework classes, implementation of framework interfaces, use of framework specific data types/classes, and use of framework specific functions/methods • framework provides specialized versions of native language data types • framework has a "large" programming interface (API) • framework use may depend on many libraries	Components under the framework are: • bound dynamically at run time by use of language annotations/dependency injection techniques (inversion of control software design pattern) • loosely coupled and largely independent of the framework • convention over configuration: developers only specify unconventional details in code as defaults are otherwise assumed • framework uses native language data types • framework has a "small" programming interface (API)

With respect to OMS3, its lightweight non invasive approach is following described.

OMS3 introduces programming language annotations as innovative methodology for describing component metadata. Instead of developing against traditional framework APIs, a component developer accommodates OMS3 annotations on component elements (e.g. classes, methods, fields) that are relevant for building the modeling solution. The framework captures component annotations through runtime introspection, and interprets related information for properly building model metadata. This simple and effective design fully adheres to the Inversion of Control principle (Fowler (2004)): as a result of model metadata, the framework drives simulation execution and data flow. This approach allows for annotating any POJOs and legacy software applications, which can be used from within the framework consequently.

This is a lightweight non-invasive approach since no framework specific data types need to be used and no framework interfaces or abstract classes need to be implemented or extended from within the model component.

Richardson (2006) demonstrated the effectiveness of this approach on other domain specific applications such as web application and enterprise frameworks.

Each and every OMS3 annotation start with the `<at>` symbol (`@`). To summarize the main functionalities, three different groups are identified (David et al. (2013)):

1. *Mandatory annotations* - These annotations are mandatory for executing a modeling component. `@Execute` is accommodated above the method signature that rules the component execution and is invoked by the framework at

runtime. @In and @Out are located above each input/output field declaration respectively. Annotated fields identify the incoming and outcoming flow of data in and to a component and they must be declared public to be accessed by the framework. OMS3 is in charge of handling the data transfer protocol between components. It exchanges any data type independently from data semantic or structure.

2. *Supportiving annotations* – These annotations are optional. However they facilitate and bound model execution by introducing additional information such as @Unit to describe the physical unit of a model parameter, or @Range to constrain the admissible value of a model parameter.

3. *Documentation annotations* – Although these annotations are optional as well, they provide useful metadata which facilitate model component readability and maintainability consequently. This information are also parsed by the framework to automatically generate model documentation (@Description, @Author, @Version).

Listing 4.1 shows a simple example of an annotated OMS-compliant component.

Listing 4.1: Example of a POJO class turned into OMS-compliant component by accommodating OMS3 annotations.

```java
package example;

import oms3.annotations.*;

@Description("Compute cylinder volume")
@Author(name = "Francesco Serafin",
        contact = "francesco.serafin.3@gmail.com")
@Keywords("cylinder volume")
@Bibliography("David, O., Ascough II, J. C., Lloyd, W.,
    Green, T. R., Rojas, K. W., Leavesley,
    G. H., & Ahuja, L. R. (2013).
    A software engineering perspective on environmental modeling
    framework design: The Object Modeling System.
    Environmental Modelling & Software, 39, 201-213.")
@VersionInfo("0.1")
@Status(Status.TESTED)
public class CylinderVolume {

  @Description("Radius of cylinder base")
  @Role(Role.PARAMETER+Role.VARIABLE)
  @Unit("meters")
  @In
  public double radius;

  @In
  public double height;

  @Out
  public double volume;

  @Execute
  public void compute() {
    volume = circleArea(radius) * height;
    System.out.println(``The volume is: `` + volume);
  }

  private double circleArea(double radius) {
    return Math.pow(radius, 2) * Math.PI;
  }

}
```

Line 3 of Listing 4.1 shows the only dependency between framework and OMS-compliant component: **import** oms3.annotations.* is required to make use of all the available OMS3 annotations.

OMS3 supports the Initialize/Run/Finalize cycle by providing two further annotations @Initialize and @Finalize along with @Execute to rule the model execution flow. This allows for tagging two extra methods within a Java class that are invoked before (@Initialize) and after (@Finalize) the main method tagged with @Execute (Figure 71).

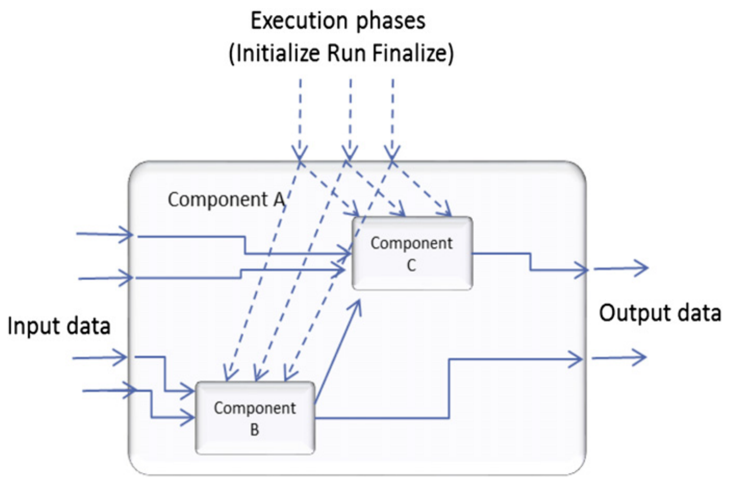


Figure 71: Execution phases, and data flow of OMS3 modeling solution, credit David et al. (2013)

In conclusion, definitions of framework invasiveness and code coupling are provided to introduce to the concept of Java annotations, which make OMS3 a lightweight non-invasive framework. Listing 4.1 shows the adaptation of a POJO test case into OMS-compliant component. Multi-language operability is exercised in Appendix B.

4.4.2.1.3 Simulation DSL

This subsection introduces to generic DSL definition and actual application to environmental modeling solutions. OMS3 DSL and its mapping into OMS3 code base are thoroughly analyzed. OMS3 DSL formal structure and the concept of simulation file are provided. Afterwords, each of the seven main DSL elements of an OMS3 simulation file are individually described. Simulation, Resources, Analysis, Summary Output, Model Efficiencies, and Simulation Output Strategy are introduced as independent subsections. Model element requires a different section since it is the DSL core concept.

A domain specific language is a programming language (usually declarative) designed to simplify the solution of domain-specific problems through dedicated notations and abstractions (Van Deursen et al. (2000)). A DSL is a very expressive and powerful but small language which works on top of a General Purpose Programming Language (GPL) with the main goal of narrowing GPL scope (Fowler (2010); Van Deursen et al. (2000)).

Regarding EMFs specific applications, a DSL interposes between the collection of model components and the modeling framework. It simplifies the definition of a set of framework instructions which are required to properly execute a modeling solution.

Traditionally, properly balanced mix of DSL and GPL facilitate modelers work and improve software application effectiveness (David et al. (2012)). Furthermore, a DSL is more flexible and powerful option to complex Graphical User Interface (GUI).

OMS3 provides a flexible and user-friendly DSL to facilitate modelers work of setting up the modeling simulation. Through the DSL, a user generates a runtime system of rules (protocol) to:

1. select the type of OMS3 simulation to run;
2. identify and list the model components required to build the modeling solution;
3. specify the mandatory input data entry points to initially feed the modeling solution pipeline;
4. describe modeling components interconnections.

OMS3 provides several DSL concepts for various purposes, which facilitate the transitioning of modeler creativity and ideas to lower level framework implementation of a modeling solution. Some examples of DSL concepts are basic model simulation, parameter calibration, sensitivity analysis, and ensemble streamflow prediction (ESP) (David et al. (2012)).
OMS3 leverages Groovy language and its provided DSL *builder design pattern* to set up a runtime simulation (Gamma (1995); Dearle (2010); David et al. (2013)).
After this brief introduction to DSL definition and the generic OMS3 usage, full description of OMS3 DSL is provided. Thus, following simulation DSL always refers to specific OMS3 application.

Independently of the simulation type, a simulation DSL file is expected to have `*.sim`, `*.luca`, `*.esp`, `*.fast`, or `*.ps` extension and adhere to the formal structure in Listing 4.2.

Listing 4.2: OMS3 modeling solution formal structure.

```
// comment
<root element>(<key:value>, <key:value>, ...) {

  <element>(<properties like above>) {
    // more subelements..
  }

  <element>(<properties like above>) {
    // more sublements or just elements with value
    <element> <value>
  }

  <element>(<properties like above>)

  <element> {
    <element ..
  }
  // more subelements
}
```

Comments can be singled lined ('// ...') or can span multiple lines ('/* ...*/') such as in C++, Java, or Groovy. There is only one root *element*, which is usually the identified simulation type. Every element might have properties, provided in parenthesis after the element name (parenthesis can be omitted if there are no properties), and sub elements within curly brackets (curly brackets can be omitted if there are no sub elements). This hierarchical structure (element, sub element) is similar to XML style but (1) results in less verbose statements, (2) can potentially contain GPL constructs (because of the Groovy language), and (3) is executable through the OMS3 runtime. *Properties* are a list of comma separated tuple of `<key>:<value>` pairs.

A basic simulation (`*.sim`) is the standard methodology to setup and run a modeling solution. Listing 4.3 shows a typical simulation DSL (Rigon et al. (2016); Bancheri (2017)).

Listing 4.3: Example of OMS3 modeling solution, credit Rigon et al. (2016) and Bancheri (2017).

```
 1  import static oms3.SimBuilder.instance as OMS3
 2  def home = oms_prj
 3  def startDate= "1994-01-01 00:00"
 4  def endDate= "1995-01-01 00:00"
 5  OMS3.sim(name:"TT_integrator") {
 6    resource "$oms_prj/lib"
 7    build(targets:"all")
 8    model(while:"reader_data_Qtt.doProcess") {
 9      components {
10
11        "reader_data_Qtt" "org.[...].OmsTimeSeriesIteratorReader"
12        "integrator"      "integrator.InjectionTimeIntegration"
13        "writer_Qint" "org.[...].OmsTimeSeriesIteratorWriter"
14
15      }
16
17      parameter{
18
19        "reader_data_Qtt.file"             "${home}/data/Qtt.csv"
20        "reader_data_Qtt.idfield"           "ID"
21        "reader_data_Qtt.tStart"            "${startDate}"
22        "reader_data_Qtt.tEnd"              "${endDate}"
23        "reader_data_Qtt.tTimestep"           60
24        "reader_data_Qtt.fileNovalue"        "-9999"
25
26        "integrator.ID"    209
27        "integrator.tStartDate"    "${startDate}"
28        "integrator.tEndDate"     "${endDate}"
29
30        "writer_Qint.file" "${home}/output/Qtt_int.csv"
31        "writer_Qint.tStart" "${startDate}"
32        "writer_Qint.tTimestep" 60
33        "writer_Qint.fileNovalue" "-9999"
34
35      }
36
37      connect {
38
39        "reader_data_Qtt.outData"   "integrator.inQoutvalues"
40        "integrator.outHMQ"         "writer_Qint.inData"
41
42      }
43    }
44  }
```

Groovy runtime (`GroovyShell.java` class) is the actual engine that parses the DSL and creates the sim element of the `SimBuilder.java` class (Listing 4.3 - line 5).

The `GroovyShell.java` class is instantiated from within the `SimBuilder.java` class and, through a cascade of nested building processes, initializes the `Sim.java` object. During the nested building processes, `GroovyShell.java` object populates `Resource`, `Build`, and `Model` (Listing 4.3 - line 6:8) fields declared in the `Sim` object.

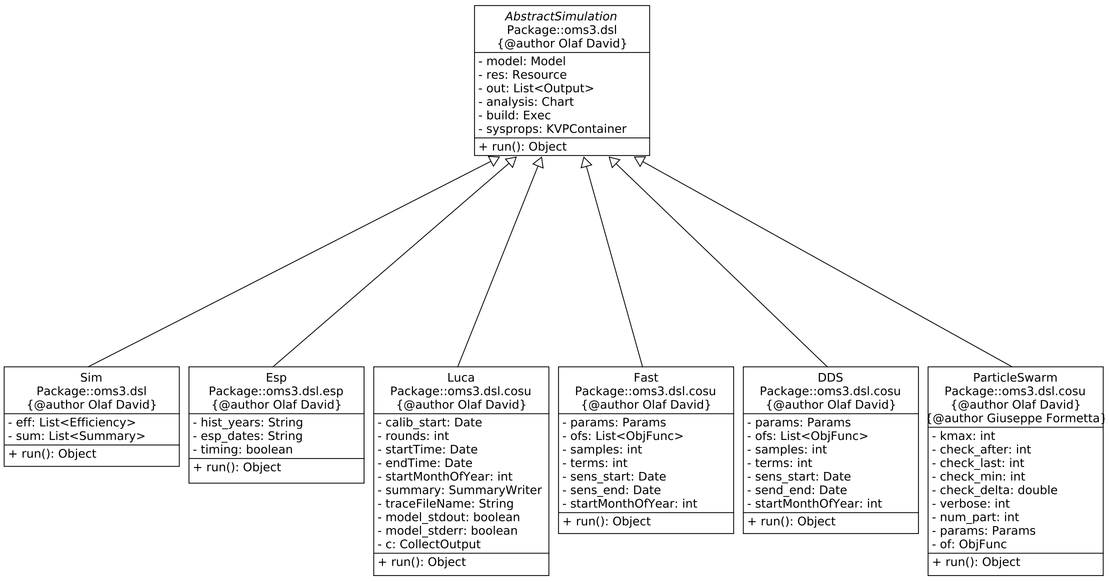


Figure 72: UML of OMS3 available modeling simulation types.

`Sim` DSL root element allows for setting up six contemporary elements: `model{}`, `outputstrategy{}`, `resource{}`, `efficiency{}`, `summary{}`, and `analysis{}`. This elements correspond to framework Java Objects, and are built by the Groovy runtime. Listing 4.3 exercises only `resource{}`, `build{}`, and `model{}`. Nevertheless, a brief introduction to every element is provided as well. `model{}` element is deeply investigated in the Modeling concepts section.

Simulation (sim) - `oms3.dsl.Sim.java`

`Sim` DSL root element is mapped into `Sim.java` class, which extends the abstract class `AbstractSimulation.java`. OMS3 provides six different types of actual implemented simulations through subclassing: `Sim`, `Esp`, `Luca`, `Fast`, `DDS`, and `ParticleSwarm` (Figure 72).

They all expose a similar API due to the subclassing behaviour but have different simulation targets. However, thanks to a thorough and effective architectural design, core parts of the framework implementation such as implicit parallelization are reused. Consequently, this dissertation describes and step through `Sim` class only.

Resources (resource) - `oms3.dsl.Resource.java`

Every simulation manages resources such as a model executable, DLLs, parameter files, climate data input, documentation, etc. The resource element allows for listing of those resources. Three different usage options of the resource listings are available:

- All jar files listed in a resource element are added to the classpath for JAVA model execution. Jar files can be referenced as local files or URLs, if the model is loaded from a remote location. If no Jar files are listed, the framework looks for model applications int the default classpath.

- All files regardless of which type are used for digest computation to ensure comprehensive hashing of all simulation resources.

- Other tools for remote execution within a cluster can use the resource listing to copy those files to other machines.

A resource section of a modeling simulation might look like Listing 4.4.

Listing 4.4: Example of resource section in OMS3 modeling simulation with list of single resources.

```
import static oms3.SimBuilder.instance as OMS3
def home = oms_prj
def startDate= "1994-01-01 00:00"
def endDate= "1995-01-01 00:00"
OMS3.sim(name:"TT_integrator") {
  resource "\$oms_prj/lib/jgt-grass-0.7.7-SNAPSHOT.jar"
  resource "\$oms_prj/lib/TT.jar"
  resource "\$oms_prj/data/ETtt.csv"
}
```

The resource values always follow the resource keyword. It also shows the use of string replacement in order to reference a common root directory (`\$oms_prj`). Alternatively the files above can be provided as a list of Strings to one resource element (Note the required brackets and parentheses - Listing 4.5). Both notations do have the same semantics.

Listing 4.5: Example of resource section in OMS3 modeling simulation with array of resources.

```
import static oms3.SimBuilder.instance as OMS3
def home = oms_prj
def startDate= "1994-01-01 00:00"
def endDate= "1995-01-01 00:00"
OMS3.sim(name:"TT_integrator") {
  resource (["$oms_prj/lib/jgt-grass-0.7.7-SNAPSHOT.jar",
             "$oms_prj/lib/TT.jar",
             "$oms_prj/data/ETtt.csv"])
}
```

Analysis (analysis) - `oms3.dsl.analysis.Chart.java`

An analysis elements provides for post run analysis by means of plotting/graphing features. It is an optional step of a modeling simulation. OMS3 provides the following types of analysis plots:

- Time series plots;
- Flow duration plots;
- Scatter plots.

The following type is specifically developed for Ensemble Streamflow Prediction (ESP):

- Esp trace analysis plots.

An analysis element can potentially contain any number of previously defined plot objects as sub-elements (Listing 4.6).

Listing 4.6: Example of available options in the OMS3 plot sub-element.

```
import static oms3.SimBuilder.instance as OMS3
OMS3.sim(name:"Efcarson") {
  \dots
  analysis(title:"Simulation Output") {
    timeseries(title:"East Fork Carson") {
      x(file:"%last/out1.csv", column:"date")
      y(file:"%last/out1.csv", column:"basin_cfs")
      y(file:"%last/out1.csv", column:"runoff[0]")
    }
    timeseries(title:"Error") {
      x(file:"%last/out1.csv", column:"date")
      calc(eq:"sim - obs") {
        sim(file:"%last/out1.csv", column:"basin_cfs")
        obs(file:"%last/out1.csv", column:"runoff[0]")
      }
      calc(eq:"sim - obs", acc:true) {
        sim(file:"%last/out1.csv", column:"basin_cfs")
        obs(file:"%last/out1.csv", column:"runoff[0]")
      }
    }
    flowduration {
      y(file:"%last/out1.csv", column:"basin_cfs")
      y(file:"%last/out1.csv", column:"runoff[0]")
    }
    scatter {
      x(file:"%last/out1.csv", column:"basin_cfs")
      y(file:"%last/out1.csv", column:"runoff[0]")
    }
  }
  \dots
}
```

Every plot object reads in datasets that are stored as CSV tabular data. A column of the CSV file is identified by (i) file name, (ii) table name, and (iii) column name (Listing 4.7). However, the analysis element can handle some shortcuts in context of the simulation. There are some examples:

Listing 4.7: OMS3 plot element.

```
x(file:"$oms_prj/SIM/0003/out1.csv", table"efc", column:"runoff")
```

A column is fully referenced with file, table, and column name. The file name is accessed with the absolute path (Listing 4.8).

Listing 4.8: OMS3 plot element.

```
x(file:"$oms_prj/SIM/%last/out1.csv", table"efc", column:"runoff")
```

A column is fully referenced with file, table, and column name. The file name is accessed with the absolute path but refers to the last simulation run. The meaning of 'last' depends on the chosen output strategy (Listing 4.9).

Listing 4.9: OMS3 plot element.

```
x(file:"%last/out1.csv", table"efc", column:"runoff")
```

In Listing 4.10, the file reference is in the simulation context. It points to a file in the last output folder for the running simulation.

Listing 4.10: OMS3 plot element.

```
x(file:"%last/out1.csv", column:"runoff")
```

If the table name is not provided, analysis element assumes a table with the simulation name (Listing 4.10). This option provides the highest flexibility in terms of independent data path referencing.

In case of n-simulation context run, OMS3 provides three predefined variables to access CSV data files.

- `%first` - The first simulation output in the run sequence. (For numbered outputs is the folder with the lowest number, for timed output the oldest simulation time);
- `%previous` - The previous simulation output in the run sequence. (For numbered output this is the folder with the highest number - 1, for timed output the second recent simulation time);
- `%last` - The last simulation output in the sequence (For numbered output this is the folder with the highest number, for timed output the most recent simulation time).

In case of a `SIMPLE` output strategy, `%first`, `%previous`, and `%last` refer to the same output folder.

Implementing those variables in a modeling simulation file has several benefits. Once a generic analysis configuration is created using `%last` variable, it always refers to the most recent output of each simulation run.

Additionally, user can always compare last and previous output runs and analyse the impact of model parameter changes.

In a further scenario, a modeler can compare the output of the last simulation run against a baseline dataset. The latter is referenced with a full qualified absolute path name.

`%first1`, `%previous`, and `%last` variables facilitate modeler's workflow by avoiding manual changes of hard coded absolute paths to analysis file at each simulation run.

Summary Output (summary) - **`oms3.dsl.Summary.java`**

The summary element provides ad-hoc statistics for selected model (state) variables. Statistical moments are computed over a selected aggregation period of time. Five aggregation periods are provided: daily, weekly, monthly, yearly, or the entire simulation. The summary of only one variable at a time can be analyzed, and the identified variable has to be tagged as model component output with `@Out` annotation.

Listing 4.11 shows a usage example of the summary element within the `SimpleModel` simulation.

Listing 4.11: Example of OMS3 summary element with single statistic.

```
sim(name:"SimpleModel") {

  // define the model
  model(classname:"tw.Thornthwaite") {
    ...
  }
  summary(time:"time", var:"basin_ro", statistics:MAX, file:stats.txt)
}
```

Here, the maximum value of the output variable `basin_ro` is computed over the total simulation time, and stored in the file stats.txt. The latter is automatically saved in the simulation output folder.

Listing 4.12: Example of OMS3 summary element with multiple statistics on a specified period of time.

```
sim(name:"SimpleModel") {

  // define the model
  model(classname:"tw.Thornthwaite") {
    \dots
  }
  summary(time:"time", var:"runoff[4]",
          statistics:MEAN+MIN+LAG1, period:YEARLY)
}
```

Listing 4.12 shows a more complex usage example: element 4 of the runoff array is aggregated over one year time and its minimum, mean and autocorrelation consequently redirected to the console standard output.

Table 5 shows the list of OMS3 provided statistical moments.

Table 5: List of OMS3 provided statistical moments.

Moment	Description
MEAN	$MEAN = \frac{1}{N}\sum_{i=1}^{N} x_i$
MAX	$MAX = max_i(x_i)$
MIN	$MIN = min_i(x_i)$
COUNT	$COUNT = count(x_i)$
RANGE	$RANGE = max_i(x_i) - min_i(x_i)$
MEDIAN	$MED = \begin{cases} Y_{(N+1)/2}, & \text{if N ia odd,} \\ \frac{1}{2}(Y_{N/2} + Y_{1+N/2}), & \text{if N is even.} \end{cases}$
STDDEV	$SD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$
VAR	$VAR = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2$
MEANDEV	$MD = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2$
SUM	$SUM = \sum_{i=1}^{N} x_i$
PROD	$PROD = \prod_{i=1}^{N} x_i$
Q1	First quartile
Q2	Second quartile
Q3	Third quartile
LAG1	LAG-1 autocorrelation

Model Efficiencies (efficiency) - `oms3.dsl.Efficiency.java`

Model efficiencies are commonly used to quantify prediction performance of a simulation model by computing parameter aggregation based on observed and simulated values of the same model property. OMS3 provides several model efficiencies. A comprehensive list is provided in Table 6.

Table 6: List of OMS3 provided model efficiencies.

Moment	Description	Equation
ABSDIF	Absolute difference	$\sum_{i=1}^{n} \lvert Q_{i,o} - Q_{i,s} \rvert$
ABSDIFLOG	Absolute difference Log	$\sum_{i=1}^{n} \lvert \ln Q_{i,o} - \ln Q_i, s \rvert$
AVE	Absolute Volume Error	$\lvert \sum_{i=1}^{n} Q_{i,s} - Q_{i,o} \rvert$
IOA	Index of Agreement	$1 - \frac{\sum_{i=1}^{n} \lvert Q_{i,o} - Q_{i,s} \rvert}{\sum_{i=1}^{n} \lvert Q_{i,s} - \overline{Q_o} \rvert + \lvert Q_{i,o} - \overline{Q_o} \rvert}$
IOA2	Index of Agreement (Pow 2)	$1 - \frac{\sum_{i=1}^{n} (Q_{i,o} - Q_{i,s})^2}{\sum_{i=1}^{n} (Q_{i,s} - \overline{Q_o})^2 + (Q_{i,o} - \overline{Q_o})^2}$
NS	Nash-Sutcliffe	$1 - \frac{\sum_{i=1}^{n} (Q_{i,o} - Q_{i,s})^2}{\sum_{i=1}^{n} (Q_{i,o} - \overline{Q_o})^2}$
NSLOG	Log of Nash-Sutcliffe	$1 - \frac{\sum_{i=1}^{n} \lvert Q_{i,o} - Q_{i,s} \rvert}{\sum_{i=1}^{n} \lvert Q_{i,o} - \overline{Q_o} \rvert}$
NS2LOG	Log of Nash-Sutcliffe (Pow 2)	$1 - \frac{\sum_{i=1}^{n} (\ln Q_{i,o} - \ln Q_{i,s})^2}{\sum_{i=1}^{n} (\ln Q_{i,o} - \ln \overline{Q_o})^2}$
BIAS	Bias	$\frac{\sum_{i=1}^{n} (Q_{i,o} - Q_{i,s})}{\sum_{i=1}^{n} Q_{i,o}}$
RMSE	Root Mean Square Error	$\sqrt{\frac{1}{n} \sum_{i=1}^{n} (Q_s - Q_o)^2}$

Listing 4.13 illustrates a usage example of efficiency element in a modeling solution. OMS3 allows for computing multiple efficiencies simultaneously by combining different performance coefficients with + operator.

Listing 4.13: Example of OMS3 efficiency implementing multiple methods.

```
sim(name:"Efcarson") {
  // define the model
  model(classname:"model.PrmsDdJh") {
    // ... parameter here
  }
  efficiency(obs:"runoff[0]", sim:"basin_cfs",
             methods:NS+NS2+ABSDIF+TRMSE)
}
```

Additional table output for requested efficiencies is produced by executing a modeling simulation (Listing 4.14).

Listing 4.14: Results of OMS3 efficiency.

```
Efficiencies        ns1        ns2      absdif       trmse
runoff/basin_cfs   0.66512    0.82971   764.30044    2.44043
```

Simulation Output Strategy (outputstrategy) - `oms3.dsl.OutputDescriptor.java`

Modeling solutions results are usually stored in output files such as times series predicted runoff, sediment yield, etc. The `outputstrategy` element of a modeling simulation provides different strategies for storing modeling outputs. These strategies are consistent methodology when dealing with subsequent simulations only.
The types of supported output strategy schemes are:

- SIMPLE: The simulation creates a folder to hold the model output files. Each new simulation run will overwrite existing files with the same name. The simulation output folder is: `<output dir>/<sim name>`.
- NUMBERED: The simulation creates a new folder for each simulation run. A new simulation will not overwrite the output from the previous one. The last simulation always names the output folder with the highest number. The simulation output folder is: `<output dir>/<sim name>/<simulation run number>`.
- TIME: The simulation creates a new folder for each simulation run. A new simulation will not overwrite the output from the previous one. The last simulation always names the output folder with the simulation start time. The simulation output folder is: `<output dir>/<sim name>/<simulation start time>`

Listing 4.15 shows a usage example of an output strategy for a simple modeling solution.

Listing 4.15: Example of output strategy provided in a OMS3 modeling simulation.

```
sim(name:"SimpleModel") {

  outputstrategy(dir:"$oms_prj/out", scheme:NUMBERED)

  // define the model
  model(classname:"tw.Thornthwaite") {
    // add parameter
    parameter {
      climateFile  "$oms_prj/data/tw/climate.cst"
    }
  }
}
```

4.4.2.1.4 Modeling concept (model) - `oms3.dsl.Model.java`

This subsection introduces to OMS3 modeling concept. This is the core part of each modeling solutions. As a result, DSL and actual mapping into framework objects are presented. An OMS3 model element provides four different sub elements.

Components, parameter, connection, and logging sub elements are independently analyzed.

The model element is the core part of every simulation type. It describes the model components to be used in the modeling solution, how to manage them, and how to feed them with proper input data and components input/output connections. An OMS-compliant component can be any `Class` that contains a method tagged with `@Execute` annotation, which indicates the component execution entry point. Listing 4.16 is extracted from Listing 4.3 (Travel Time analysis from Rigon et al. (2016)) and highlights three of the most important model sub elements. The model element can manage five sub elements total: `component{}`, `parameter{}`, `connect{}`, `resource`, and `logging{}`. Listing 4.16 illustrates a usage case for `components{}`, `parameter{}`, and `connect{}` only. Nevertheless, except for resource previously described, a brief introduction to the logging sub element is performed as well.

Listing 4.16: Example of OMS3 model element extracted from Listing 4.3, credit Rigon et al. (2016) and Bancheri (2017).

```
model(while:"reader_data_Qtt.doProcess") {
  components {

    "reader_data_Qtt" "org.[...].OmsTimeSeriesIteratorReader"
    "integrator"      "integrator.InjectionTimeIntegration"
    "writer_Qint" "org.[...].OmsTimeSeriesIteratorWriter"

  }

  parameter{

    "reader_data_Qtt.file"             "${home}/data/Qtt.csv"
    "reader_data_Qtt.idfield"           "ID"
    "reader_data_Qtt.tStart"            "${startDate}"
    "reader_data_Qtt.tEnd"              "${endDate}"
    "reader_data_Qtt.tTimestep"           60
    "reader_data_Qtt.fileNovalue"       "-9999"

    "integrator.ID"    209
    "integrator.tStartDate"    "${startDate}"
    "integrator.tEndDate"     "${endDate}"

    "writer_Qint.file" "${home}/output/Qtt_int.csv"
    "writer_Qint.tStart" "${startDate}"
    "writer_Qint.tTimestep" 60
    "writer_Qint.fileNovalue" "-9999"

  }

  connect {

    "reader_data_Qtt.outData"   "integrator.inQoutvalues"
    "integrator.outHMQ"         "writer_Qint.inData"

  }
}
```

Modeling components (components)

The modeling components sub element allows for listing all the OMS-compliant components involved in the modeling solution and assigning them a component ID. This sub element adheres to the formal structure in Listing 4.17.

Listing 4.17: OMS3 component sub-element formal structure.

```
model(while:"reader_data_Qtt.doProcess") {
  components {

    // list of every component involved in the modeling solution
    <cID>    <package + classname>

  }
```

The component `<cID>` name is a user defined unique component ID which points to the actual component name preceded by the entire package name `<package + classname>`. The `<cID>` is used across the entire model element to refer to that specific model component. Both `<cID>` and `<package + classname>` are `String` objects and are space separated. Listing 4.18 shows the component sub element for modeling solution in Listing 4.16.

Listing 4.18: Example of OMS3 component sub-element.

```
model(while:"reader_data_Qtt.doProcess") {
  components {

    "reader_data_Qtt" "org.[...].OmsTimeSeriesIteratorReader"
    "integrator"      "integrator.InjectionTimeIntegration"
    "writer_Qint" "org.[...].OmsTimeSeriesIteratorWriter"

  }
```

The Groovy runtime parses the components sub element during the building process of the modeling solution, and stores `<cID>` and `<package + classname>` into a list of `<key:value>` pairs in the `Model.java` object.

Parameters (parameter)

The model parameter sub element allows the specification of input values, which are the entry points to initially feed a model simulation pipeline. Listing 4.19 illustrates the parameter sub element formal structure.

Listing 4.19: OMS3 parameter subelement formal structure.

```
  parameter(<key:value>, <key:value>, ...) {

    // list of input parameters
    <cID>.<paramName>    <value>

  }
```

`<cID>` is the component ID of modeling component that contains `<paramName>`. `<paramName>` matches the `Class` field tagged with `@In` OMS annotation.

This subsection can reference an external file that contains a list of model parameters (e.g. Listing 4.21) and additionally specify extra parameters between curly brackets (Listing 4.20).

Listing 4.20: Example of OMS3 parameter subelement.

```
model(while:"reader_data_Qtt.doProcess") {
  // parameter
  parameter(file:"params.csd") {
    climateFile  "c:/projects/ngmf.models/src/tw/climate.cst"
    outputFile   "output.csv"
    runoffFactor 0.5
    latitude     35.0
```

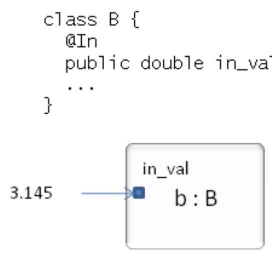
```
      smcap         200.0
    }
  }
```

Listing 4.21: Example of OMS3 parameter file.

```
@S,Parameter,
  created by, Dr. Bancheri,
  created at,Thu Jun 25 13:44:42 MDT 2017,


@P, alfa_r,1.131

@P, alfa_s,0.455

@P, meltingTemperature,2.0298

@P,combinedMeltingFactor,0.160

@P,freezingFactor,0.0035

@P,radiationFactor,8.52502271122546E-5

@P,alfa_l,0.4106

@P,kc_canopy_out,0.21

 @P,s_RootZoneMax,144.98
```

The parameter `file` key takes a file name as paired value (`params.csd`) (Listing 4.20 - line 3). File name can be referenced with absolute or relative file path, that relates to the base directory of the OMS project.

The parameter sub element can also contains a list of `<cID>.<paramName> <value>` pairs (see Listing 4.20). Values are space separated from their keys and have valid Java/Groovy data types such as Strings, Numbers, Files, etc. Those data types have to match the data type of the corresponding `@In` field in the model component. However, OMS3 SPI system attempts to convert the values into the proper original component field data type, if the value is specified as generic String. For example, the climateFile value (Listing 4.20 - line 4) is provided as String input parameter. Since climateFile original data type is a File object, OMS3 SPI converts converts the String input and instantiate a new File. Figure 73 illustrates the actual implementation of the parameter DSL principle.



```
class B {
  @In
  public double in_val;
  ...
}
```

Figure 73: Usage example of @In annotation, credit David et al. (2013).

If file property and parameter values are provided and specify the same parameter, the sub element will overwrite the parameter values specified in the file property (Listing 4.20).

Listing 4.22 shows an additional use of the parameter element. Multiple parameter elements can help splitting parameter sets in groups and allow for redefinition.

Listing 4.22: Usage example of parameter subelement in OMS3 modeling simulation.

```
model(classname:"my.model") {
  // parameter defintion
  parameter(file:"params.csv")           // parameterfile only
  parameter(file:"params-dates.csv")     // parameterfile only
  parameter(file:"params-files.csv") {   // parameterfile+explicit
    testdir "/tmp/test"
  }
  parameter {                            // only explicit parameter
    coeff 2.34
  }
}
```

More generally, parameter value reading and setting order works as follow: a parameter at a higher line number in the sim file overwrites the same one at a lower line number. It is not relevant if it comes from a file or is explicitly specified. The Groovy runtime parses the parameter sub element during the building process, and stores `<cID>.<paramName>` and `<value>` into a `Param` Object. Eventually, a list of `Param` Objects gets created as a field of the main `Model` object.

Component connections (connect)

The connect sub element is a required section to define connections of several modeling components into a modeling solution. It adheres to the formal structure in Listing 4.23.

Listing 4.23: OMS3 connect subelement formal structure.

```
  connect {

    // @Out -> @In
    <cID_a>.<outVar>    <cID_b>.<inVar>
    <cID_a>.<outVar>    <cID_c>.<inVar>

  }
```

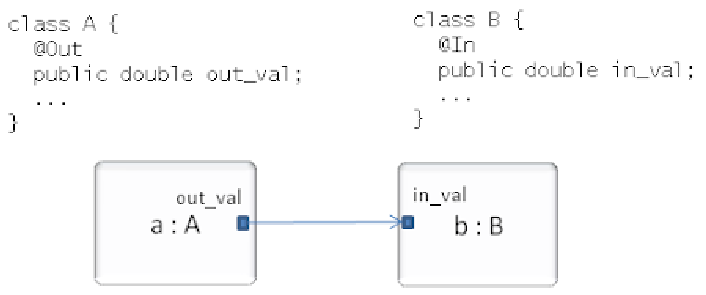
`<outVar>` and `<inVar>` match the original model component `Class` fields tagged with `@Out` and `@In` OMS annotations respectively. They are specified as space separated `<key> <value>` pair and are fields of different component Classes (`<cID_a>` and `<cID_b>` accordingly). Figure 74 illustrates the actual implementation of the component connection DSL principle.

A `Class` field can be contemporary tagged with both `@In` and `@Out` OMS annotations. Input/output fields can be any type of Java Object. Proper connection happens when Output to Input field Objects match. If they don't match, OMS3 SPI system attempts to convert the Output Object into the required Input Object. If the conversion fails, an error message is thrown.

An actual example is shown in Listing 4.24 and extracted from Listing 4.3 (parameter section omitted for sake of brevity).

Listing 4.24: Example of OMS3 connect subelement derived from Listing 4.3.

```
model(while:"reader_data_Qtt.doProcess") {
  components {

    "reader_data_Qtt" "org.[...].OmsTimeSeriesIteratorReader"
```



```
class A {
  @Out
  public double out_val;
  ...
}
```

```
class B {
  @In
  public double in_val;
  ...
}
```

Figure 74: Usage example of `@Out` to `@In` component fields connection, credit David et al. (2013).

```
      "integrator"        "integrator.InjectionTimeIntegration"
      "writer_Qint"       "org.[...].OmsTimeSeriesIteratorWriter"

    }

    ...

    connect {

      "reader_data_Qtt.outData"  "integrator.inQoutvalues"
      "integrator.outHMQ"        "writer_Qint.inData"

    }
```

Here, `outData` of the component `reader_data_Qtt` feeds the input variable `inQoutvalues` required by integrator component. `outHMQ` of integrator then feeds `inData` variable and solves the dependency with `writer_Qint` component.

The Groovy runtime parses the connect sub element during the building process, and stores `<cID_a>.<outVal>` and `<cID_b>.<inVal>` into a list of `<key:value>` pairs `Objects`. This is the `out2in` field of the main `Model.java` object for component `<cID_a>`.

Component connections happen at runtime: when `<cID_a>` simulation concludes the OMS controller transfers `Object` reference pointing to `<outVal>` memory space to `<inVal>` of component `<cID_b>`.

Logging (logging)

The logging sub-element is an optional part of a model element. It controls the logging levels for single components or for the whole model. In order to use the logging feature, components have to obtain and use OMS3 logger accordingly.

A logger is an object that allows output handling based on logging levels. Such levels usually indicate the severeness of a message. The Java logging infrastructure supports 7 logging levels by default, ranging from `FINEST` (the lowest priority or importance) to `SEVERE` (the highest importance). In addition, OMS3 provides level `OFF` to completely turn off every logging message. If a logging level is provided, every logging message of identical or higher priority are sent and printed to console standard output.

Listing 4.25 illustrates logging usage example. Here, the logging element is part of the model element. It lists the component class names and associates them with dedicated log levels for a simulation run.

Listing 4.25: Example of OMS3 logging subelement with logging level per component.

```
model(classname:"my.model") {
  // logging definition
  logging {
    "StreamFlow" "INFO"
    "GwFlow"     "CONFIG"
  }
}
```

The component `StreamFlow` in `my.model` is assigned the logging level `INFO`, the `GwFlow` component is assigned a finer grained `CONFIG` logging level. The default logging level for every other mode component is set to `WARNING` by default.

Listing 4.26: Example of OMS3 logging subelement with one dedicated component logging in addition to generic logging level.

```
model(classname:"my.model") {
  // logging definition
  logging (all:"INFO"){
    "StreamFlow" "FINEST"
  }
}
```

Listing 4.26 shows that the default logging level for each model component is set to `INFO`, while `StreamFlow` is assigned the most verbose logging level.

Listing 4.27: Example of OMS3 logging subelement with generic logging level.

```
model(classname:"my.model") {
  // logging definition
  logging (all:"OFF")
}
```

The logging element in Listing 4.27 turns off the logging system for the whole model. Completely disabling the logging system means that even severe problems within modeling components are not reported.

The modeling component has to be properly setup to accommodate OMS3 logging functionalities (Listing 4.28). This allows for leveraging the previously described logging features within a modeling solution. The modeling component has to import Java logging utility and implement static reference of a logger object in `Class` field declaration (Listing 4.28 – line 6).

Listing 4.28: Implementation of Logging class in OMS-compliant component.

```
1  import java.util.logging.*;                          // 1.
2    ...
3    public class Ddsolrad {
4
5    static final Logger log =
6       Logger.getLogger("oms3.model." +
7                        this.class.getSimpleName()); //2.
8    ...
9
10   @Execute
11   public void exec() {
12     ...
13
14     if (log.isLoggable(Level.INFO)) {               // 3.
15       log.info("Solrad " + basin_potsw);            // 4.
16     }
17   }
18   ...
19 }
```

Summarizing, to implement the `Logging` feature, four main steps are followed (Listing 4.28):

1. Import the logging classes from the `java.util` package.

2. Obtain a logger instance using the `Logger.getLogger()` call. Declare this reference static to share it across all instances of this class and final to make it a constant. The argument must start with the String `oms3.model.` and must end with the component's simple class name. Use `getSimpleName()` as shown in Listing 4.28 to obtain this name from the class itself, rather than hard typing it to the logger as a String.

3. The logger can be used at any location within the component methods. Listing 4.28 shows a *guarded logging* as recommended practice. This pattern checks if a logging statement results in a logging output before it gets executed. This reduces memory fragmentation and reduces garbage collection by avoiding creation of unnecessary strings if logging levels are disabled. Statement in Listing 4.28 checks if logging system enables `INFO` and higher levels.

4. The statement issues the logging message at the `INFO` level. Use the methods `severe()`, `warning()`, `info()`, `config()`, `fine()`, `finer()`, and `finest()` accordingly.

The use of Logging system in modeling components provides for high flexibility of diagnostics and messaging, and is efficiently configurable from within a modeling simulation.

4.4.2.1.5 Simulation run

This subsection introduces to framework workflow and analyzes a sample exercise of a model simulation run.

As previously stated, Groovy runtime parses the entire `*.sim` file and builds every object required by the modeling solution. The final simulation Object is then ready for starting the overall computation.

Through a reflective call, OMS3 `SimBuilder` invokes the `run()` method implemented in the `AbstractSimulation.java` class (Figure 72) and consequently overridden in every Class that extends it. This dissertation describes the run method of a `Sim.java` Object.

Listing 4.29: OMS3 invoke method in modeling simulation.

```java
/**
 * Invokes a simulation method. (run | doc | analysis | ...)
 *

 * @author Olaf David
 * @param target the target simulation object
 * @param name the name of the method (e.g. run())
 * @throws Exception generic exception
 */
private static Object invoke(Object target,
                             String name) throws Exception {
  return target.getClass().getMethod(name).invoke(target);
}
```

Initially, the **`super`**`.run()` method is called to initialize the simulation run by setting up system properties if provided, and checking if the `Model` Object has been created.

Listing 4.30: OMS3 run method.

```java
/**
 *
 * @author Olaf David
 */
protected void initRun() {
  setSystemProperties();
  if (getModelElement() == null) {
    throw new ComponentException("missing 'model' element.");
  }
}

/**
 *
 * @author Olaf David
 */
public Object run() throws Exception {
  initRun();
  return null;
}
```

Then, the model component is retrieved from the `Model` Object and the methods tagged with `@Initialize` OMS3 annotation are invoked through a reflective call.

Listing 4.31: OMS3 initialize reflective call.

```java
// @author Olaf David
ComponentAccess.callAnnotated(comp, Initialize.class, true);
```

This initializes the model components and executes algorithms implemented in component methods tagged with `@Initialize`.

The modeling simulation pipeline is fed with input parameters before invoking the `@Execution` method.

Listing 4.32: OMS3 input parameter read in and set up.

```java
// @author Olaf David
// setting the input data;
UnifiedParams parameter = model.getParameter();
boolean success = parameter.setInputData(comp, log);
```

The most important step in a simulation run is the execution of the core part of the entire modeling solution. This is achieved by invoking the methods tagged with `@Execute` annotation.

Listing 4.33: OMS3 execute reflective call.

```java
// @author Olaf David
ComponentAccess.callAnnotated(comp, Execute.class, false);
```

When the core computation is over, methods tagged with the `@Finalize` annotation are invoked to conclude the modeling solution run.

Listing 4.34: OMS3 finalize reflective call.

```java
// @author Olaf David
ComponentAccess.callAnnotated(comp, Finalize.class, true);
```

The `ComponentAccess` class manages reflective accesses to components internals. As a result, it allows for final component integration into a modeling solution. More specifically speaking, the `callAnnotated` method extracts the method of interest with a reflective call and invokes it.

Listing 4.35: OMS3 generic implementation of `callAnnotated` method.

```java
/**
 * Call an method by Annotation.
 *
 * @author Olaf David
 * @param o the object to call.
 * @param ann the annotation
 * @param lazy if true, the a missing annotation is OK. if false the
 * annotation has to be present or a Runtime exception is thrown.
 */
public static void callAnnotated(Object o,
                   Class<? extends Annotation> ann, boolean lazy) {
  try {
    getMethodOfInterest(o, ann).invoke(o);
  } catch (IllegalAccessException ex) {
    throw new RuntimeException(ex);
  } catch (InvocationTargetException ex) {
    throw new RuntimeException(ex.getCause());
  } catch (IllegalArgumentException ex) {
    if (!lazy) {
      throw new RuntimeException(ex.getMessage());
    }
  }
}
```

The overall execution of the entire modeling solution is managed by the `Controller.java` class. The next subsection introduces to OMS3 `While` conditional implemented in simulation in Listing 4.3 line 8 and the `internalExec()` method of the `Controller.java` class. `internalExec()` method is of specific importance since it fires up, executes, and shuts down Java concurrent threads implicitly required by the modeling solution.

4.4.2.1.6 Controller Class and Implicit parallelism

This subsection describes the core part of OMS3 model simulation run and introduces to OMS3 implicit parallelism.

The `While` conditional Class works as an OMS compliant component since its `execute()` method is tagged with the `@Execute` annotation.

Listing 4.36: OMS3 **`while`** conditional execution.

```java
/**
 * While Component.
 *
 * @author Olaf David
 *
 */
public class While extends Conditional {

  @Override
  @Execute
  public void execute() throws ComponentException {
    check();
    while (cond.alive) {
      internalExec();
    }
  }
}
```

This OMS component is bundled into the modeling solution when the Groovy runtime parses the model element of the `*.sim` file.

This component repeatedly calls the `internalExec()` method while the condition is true (or alive) `cond.alive`.

The `internalExec()` method is the most relevant method of the `Controller.java` class. It retrieves the collection of modeling components comps and initializes the counter of the number of threads to `comps.size()`.

Listing 4.37: OMS3 implicit parallelization.

```java
/**
 *
 * @author Olaf David
 *
 */
protected void internalExec() throws ComponentException {
  Collection<ComponentAccess> comps = oMap.values();
  [...]
  latch.load(comps.size());
  final ExecutorService executor = getExecutorService();
  if (rc == null) {
    rc = new Runnable[comps.size()];
    int i = 0;
    for (final ComponentAccess co : comps) {
      rc[i++] = new Runnable() {

        @Override
        public void run() {
          try {
            co.exec();
            latch.countDown();
          } catch (ComponentException ce) {
            synchronized (lock) {
              if (E == null) {
                E = ce;
              }
            }
            latch.open();
            Threads.shutdownAndAwaitTermination(executor);
          }
        }
      };
    }
  }
  if (E == null) {
    for (Runnable r : rc) {
      executor.submit(r);
    }
  }

  try {
    latch.await();
  } catch (InterruptedException IE) {
    // nothing to do here.
  }
  [...]
 }
```

Then, a vector of `Runnable` of length `comps.size()` is allocated and initialized, so each component has its own `Runnable` Object. During the initialization phase, the `run()` method of the `Runnable` interface is implemented to be executed by its dedicated thread. The `run()` method simply calls the component execution and decreases the latch counter when the simulation is over. The implementation of the `Latch` class is available at Listing 4.38.

Listing 4.38: OMS3 implicit parallelization.

```
/**
 *
 * @author Olaf David
 *
 */
static private class Latch {

  private int count;
  private final Lock lock = new ReentrantLock();
  private Condition condition = lock.newCondition();

  void load(int count) {
    this.count = count;
  }

  void open() {
    lock.lock();
    try {
      count = 0;
      condition.signal();
    } finally {
      lock.unlock();
    }
  }

  void countDown() {
    lock.lock();
    try {
      if (--count <= 0) {
        condition.signal();
      }
    } finally {
      lock.unlock();
    }
  }

  void await() throws InterruptedException {
    lock.lock();
    try {
      while (count > 0) {
        condition.await();
      }
    } finally {
      lock.unlock();
    }
  }
}
```

After initialization, the `Runnable` tasks are submitted by the `ExecutorService` for execution. The `ExecutorService` manages the scheduling of component execution. Threads communicates through component interconnection: a simulation component starts when required inputs (`@In` annotated fields) are satisfied; component outputs (`@Out` annotated fields) become following connected component inputs, which executes only when every input is satisfied. This implementation is based off of data flow multi-process synchronization, and is a common application of the producer-consumer design pattern.

4.4.2.2 *Graph Modeling Structure: NET3*

NET3 is graph modeling structure implemented as additional simulation type to the already large variety provided by OMS3.

Figure 75 illustrates NET3 conceptual design. Here, a modeling framework is represented by a red rounded rectangle. In the left side of Figure 75, OMS3 orchestrates the workflow of a single modeling solution (credit Bancheri (2017)). In the right side, OMS3+NET3 enable each node of the graph data structure to manage a modeling solution, and orchestrate modeling solutions interconnected workflow.

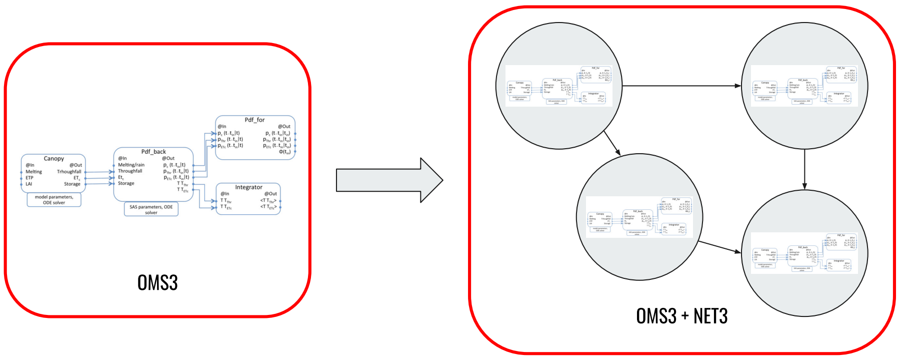


Figure 75: NET3 conceptual design. From a single OMS3 modeling solution to interconnected and intercommunicating modeling solutions.

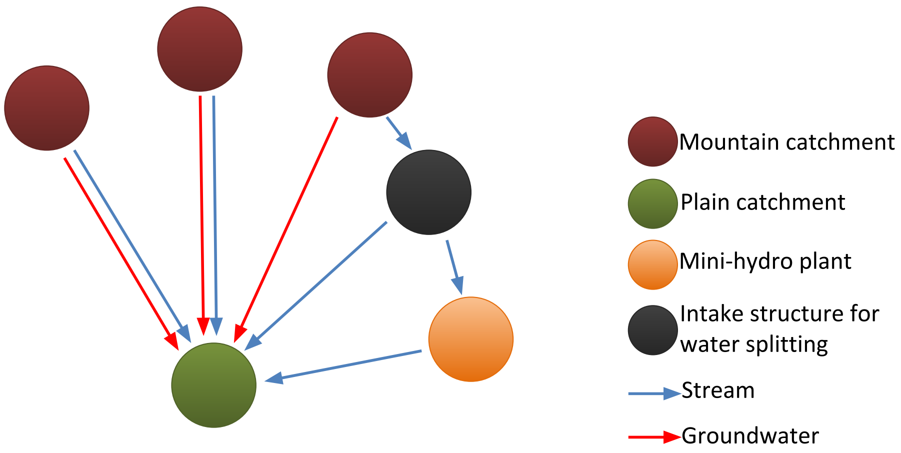


Figure 76: NET3 conceptual design. Every node of the graph modeling structure runs a different modeling solution to better fit physical processes description. Intercommunication between modeling solutions happens with unlimited number of variables.

This section is organized as follow. Paragraph *DiGraph API – `oms3.ds.graph.DiGraph.java`* introduces to NET3 graph data structure implementation and API. Paragraph *Searching algorithms* describes architectural design principles that drove the implementation of traversing algorithms and required data structures. Finally, paragraph *Graph simulation* delineates the actual integration of NET3 graph data structure and search algorithms into OMS3 workflow, by describing NET3 DSL, and its management of parallel modeling solution runs and their interconnection.

4.4.2.2.1 DiGraph API - `oms3.ds.graph.DiGraph.java`

The `DiGraph` Java class implements the graph modeling structure and exposes required API only.

UML in Figure 77 illustrates public methods exposed by `DiGraph` API, private methods, and the structure of private class `Family`. No public fields are exposes by `DiGraph` API to avoid uncontrolled changes in their state.

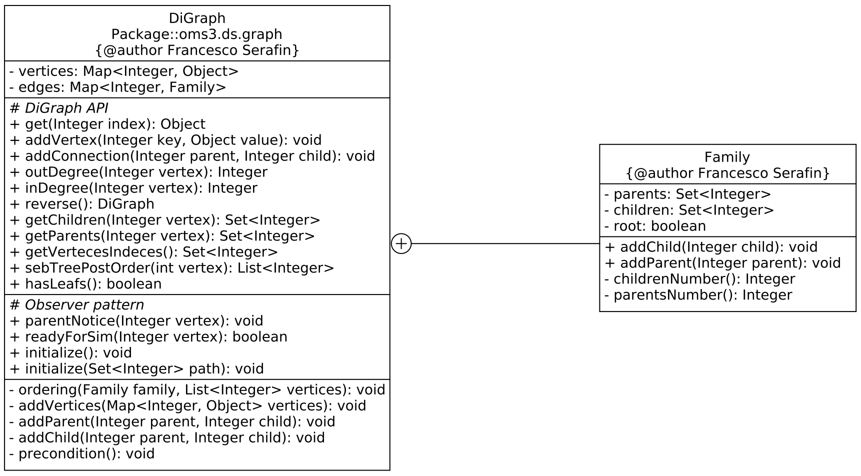

DiGraph
Package::oms3.ds.graph
{@author Francesco Serafin}
- vertices: Map<Integer, Object>
- edges: Map<Integer, Family>
# DiGraph API
+ get(Integer index): Object
+ addVertex(Integer key, Object value): void
+ addConnection(Integer parent, Integer child): void
+ outDegree(Integer vertex): Integer
+ inDegree(Integer vertex): Integer
+ reverse(): DiGraph
+ getChildren(Integer vertex): Set<Integer>
+ getParents(Integer vertex): Set<Integer>
+ getVertecesIndeces(): Set<Integer>
+ sebTreePostOrder(int vertex): List<Integer>
+ hasLeafs(): boolean
# Observer pattern
+ parentNotice(Integer vertex): void
+ readyForSim(Integer vertex): boolean
+ initialize(): void
+ initialize(Set<Integer> path): void
- ordering(Family family, List<Integer> vertices): void
- addVertices(Map<Integer, Object> vertices): void
- addParent(Integer parent, Integer child): void
- addChild(Integer parent, Integer child): void
- precondition(): void
Family
{@author Francesco Serafin}
- parents: Set<Integer>
- children: Set<Integer>
- root: boolean
+ addChild(Integer child): void
+ addParent(Integer parent): void
- childrenNumber(): Integer
- parentsNumber(): Integer

Figure 77: UML of NET3 `DiGraph` class.

The modeling structure is built upon two private fields declared as Java Map classes (Listing 4.39).

Listing 4.39: NET3 declaration of vertices and edges data structures.

```java
Map<Integer, Object> vertices;
Map<Integer, Family> edges;
```

Vertices and edges are instantiated as a `ConcurrentHashMap` from the `DiGraph` constructor (Listing 4.40).

Listing 4.40: NET3 instantiation of vertices and edges.

```java
public DiGraph() {
  vertices = new ConcurrentHashMap<>();
  edges = new ConcurrentHashMap<>();
}
```

Here, `ConcurrentHashMap` has been identified as most suitable data structure since it is a concurrent collection, and the `DiGraph` is designed to work under heavily threaded environment. The reason that motivated this choice is following briefly summarized.

A concurrent collection differs from synchronized collections such as `Vector` and `Hashtable` (which are available since first version of Java Development Kit (JDK)), or `Collections.synchronizedXxx` factory methods (which are available since JDK 1.2) (Goetz and Peierls (2006)). For the sake of example, `Collections.synchronizedMap` in Listing 4.41 is the factory method that allocates a synchronized wrapper class of the original `HashMap`.

Listing 4.41: Example of Java synchronized wrapper class.

```

Map<Inter, Object> tmpMap =
        Collections.synchronizedMap(new HashMap<Integer, Object>();
```

The difference between synchronized and concurrent collections sits in the type of synchronization policy implemented (Goetz and Peierls (2006)):

A. synchronized collections have their state encapsulated and each and every public method is synchronized. As a result, access to the collection is serialized, namely allowed to a single thread at once;

B. concurrent collections allow multiple threads to access the collection state simultaneously. The synchronization policy is based on a completely different locking mechanism: for instance, *locking striping* is implemented to make `ConcurrentHashMap` thread safe. This mechanism is based off of a fine-grain partition locking. User can define a set of locks (by default 16), each of which controls one segment of the hash buckets. As a consequence, concurrent reading threads are always guaranteed map access, also while writing threads are performing stored data modifications. Simultaneous writing threads are actually constrained to the number of guard locks defined.

Synchronization policy B definitely allows for a far higher throughput in heavily threaded environments compared to synchronization policy A. It dramatically improves scalability especially in systems with many multi-core processors (Goetz and Peierls (2006)).

Keys of the vertices map are the indices of each node of the modeling structure. The corresponding Object is used for the implicit parallelization and deeply described in subparagraph *NET3 implicit parallelization*.

The edges map stores the indices of each node of the modeling structure and corresponding Family connections. The Family private class simply stores relationships to and from the subject node. Figure 78 illustrates the concept of NET3 Family.

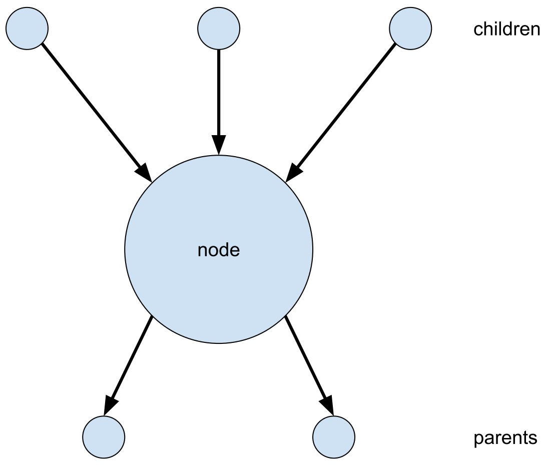


Figure 78: NET3 family concept.

Here, parents are indices of the nodes connected with outcoming edges from the subject node, children are indices of the nodes connected with incoming connections to the subject node (Figure 4.42).

Listing 4.42: NET3 Family private class.

```java
private class Family {

  private final Set<Integer> parents;
  private final Set<Integer> children;
  private boolean root = false;

  public Family() {
    parents = new HashSet<>();
    children = new HashSet<>();
  }

  public void addChild(Integer child) {
    children.add(child);
  }

  public void addParent(Integer parent) {
    if (parent == 0) {
      root = true;
      return;
    }
    parents.add(parent);
  }

  private Integer childrenNumber() {
    return children.size();
  }

  private Integer parentsNumber() {
    return parents.size();
  }
}
```

The `DiGraph` API can be split in two groups of public methods: standard directed graph API methods and observer pattern related methods (Listing 4.42). The latter are deeply described in paragraph *Graph simulation* since the observer pattern is fundamental architectural design aspect of the overall graph implicit parallel simulation run.

Standard directed graph API methods allow for building the `DiGraph` and exposing `DiGraph` characteristics for searching algorithms.

The `DiGraph` is built by Groovy runtime by parsing a topology file. The latter is a text file of space separated `<from_index_node> <to_index_node>` tuple. The entire graph is built by using two methods only: `addConnection(parent, child)`, and `addVertex(child, new HashMap<>())` (Listing 4.43).

Listing 4.43: NET3 Family private class.

```java
while ((currentLine = topology.readLine()) != null) {

  String[] family = currentLine.split(``\\s+'');

  [...]

  int parent = Integer.parseInt(family[1]);
  int child = Integer.parseInt(family[0]);


  [...]

  digraph.addConnection(parent, child);
  digraph.addVertex(child, new HashMap<>());

  [...]



}
```

The method `addVertex` simply adds the key of the node and an empty `HashMap` for later use to the vertices map (Listing 4.44).

Listing 4.44: NET3 vertex initialization.

```java
public void addVertex(Integer key, Object value) {
  if (key != 0) {
    vertices.putIfAbsent(key, value);
  }
}
```

The method `addConnection` builds the families of both parent and child: it adds parent to the child's family, and child to the parent's family (Listing 4.45).

Listing 4.45: NET3 family initialization.

```java
public void addConnection(Integer parent, Integer child) {
  addParent(parent, child);
  addChild(parent, child);
}

private void addParent(Integer parent, Integer child) {
  Family family =
    (edges.containsKey(child)) ? edges.get(child) : new Family();
  family.addParent(parent);
  edges.put(child, family);
}

private void addChild(Integer parent, Integer child) {
  if (parent == 0) {
    return;
  }
  Family family =
    (edges.containsKey(parent)) ? edges.get(parent) : new Family();
  family.addChild(child);
  edges.put(parent, family);
}
```

Other methods provided by the API such as `outDegree` and `inDegree` return the number of connections outcoming from and incoming to the source node respectively. `getChildren` and `getParents` return the set of nodes connected to the source node.

`subTreePostOrder` and reverse have been implemented for later use and are part of the architectural design aspects for implementing automatic parallel multi-site calibration (Listing 4.46 and Listing 4.47).

Listing 4.46: NET3 post ordering.

```java
public List<Integer> subTreePostOrder(int vertex) {
  List<Integer> vertices = new ArrayList<>();
  Family fam = edges.get(vertex);
  ordering(fam, vertices);
  vertices.add(vertex);
  return vertices;
}

private void ordering(Family family, List<Integer> vertices) {
    for (Integer child : family.children) {
      Family childFamily = edges.get(child);
      ordering(childFamily, vertices);
      vertices.add(child);
    }
  }
```

Listing 4.47: NET3 `DiGraph` reversing.

```
public DiGraph reverse() {
  precondition();
  DiGraph reversedDiGraph = new DiGraph();
  edges.keySet().forEach(vertex -> {
    Set<Integer> formerChildren = edges.get(vertex).children;
    formerChildren.forEach((newParent) -> {
      reversedDiGraph.addConnection(newParent, vertex);
    });
  });
  reversedDiGraph.addVerteces(vertices);
  return reversedDiGraph;
}
```

`reverse` method simply reverses the connection orders thus transforming parents in children and vice versa. It is an exact copy of the original graph with reversed connections (Listing 4.47).

4.4.2.2.2 Searching algorithms

Searching algorithms are the core part of a graph data structure. As a result, the `oms3.ds.graph.traversers` package has been accurately designed in order to allow for future development and capabilities expansion.
The architectural design principles of the searching algorithms implementation are:

1. **decoupling data representation from processing components:** this allows for decoupling actual algorithm implementations from the representation of the data structure (Sedgewick and Wayne (2011)). Consequently, each algorithm is implemented in its own class, which allows for easier maintenance and development. `DiGraph.java` is the data structure representation, deeply analyzed in the previous section, while the collection of classes in package `oms3.ds.graph.traversers` encapsulate search and traversing algorithms;

2. **algorithm encapsulation:** it results from (1) analysis of a family of algorithms, (2) identification of common implementation parts between applications, (3) separation of parts that vary in each algorithm from parts that remain identical across applications, (4) encapsulation of varying behaviours in dedicated classes (Gamma (1995); Freeman et al. (2004)). The design principle is summarized in *"Identify the aspects of your application that vary and separate them from what stays the same"* (Freeman et al. (2004)).

3. **polymorphism & dynamic binding:** polymorphic objects match same interface or abstract class but behave differently and are implemented on a IS-A relationship (Gamma (1995)). *"By programming to an interface and not an implementation"*, the inherited interface (or extended abstract class) allows programmer for declaring or implementing against common behaviour of polymorphic objects at compile time and instantiating or committing to behaviour of the actual object at runtime (Gamma (1995); Freeman et al. (2004)).

4. **composition:** complex functionalities are obtained by composing objects with HAS-A relationship (Gamma (1995); Freeman et al. (2004)). *"Favor composition over inheritance"* doesn't break software encapsulation (Gamma (1995)).

5. **dependency inversion principle:** dependencies between abstract high-level classes and concrete low-level classes have to be minimized (Freeman et al. (2004)). *"Depend upon abstractions. Do not depend upon concrete classes"* (Freeman et al. (2004)).

Two algorithms are implemented: breadth first and depth first searches. However, the flexible design allows for easily expand the searching API with additional custom algorithms.

BFS and DFS have been introduced in paragraph *DiGraph API and traversing algorithms* already. Additionally, this API implements downstream and upstream search directions for both algorithms.

Data representations of BFS and DFS are decoupled from actual algorithm implementations. Subparagraph *Data structures: Strategy Pattern – `oms3.ds.graph.traversers.AlgorithmDS.java`* shows data representation of BFS and DFS, which take advantage of the benefit of First-In-First-Out (FIFO) andLast-In-First-Out (LIFO) data structures respectively to expose one single common API. Subparagraph *Abstract implementation of searching algorithms: Factory Method Pattern – `oms3.ds.graph.traversers.GraphSearchAlgo.java`* describes the lightweight and flexible abstract implementation of searching algorithms as a result of proper data structure choice.

Data structures: Strategy Pattern – `oms3.ds.graph.traversers.AlgorithmDS.java`

In order to avoid code duplication, improve code reuse, facilitate encapsulation and separation of concerns, the main difference between BFS and DFS has been identified in their data representation. As a result, proper design of Strategy Pattern allows for encapsulating the different data representation in dedicated Java class but exposing one common identical API.

The UML in Figure 79 illustrates the Strategy Pattern implemented.

`AlgorithmDS` API exposes three methods: `add(Integer vertex)`, `delete()`, `isEmpty()`. Independently from the underneath data structure, the client interacts with:

1. `add(Integer vertex)`: a new node is added to the data structure;

2. `delete()`: a node is retrieved and removed from the data structure;

3. `isEmpty()`**:** always returns the state of the data structure.

The difference between instantiating a `BFPalgoDS` and a `DFPalgoDS` sits in the type of data structure implemented: the first class implements a FIFO data structure, the second class implements a LIFO data structure.

The `BFPalgoDS` data structure is a `Queue`, which supports the `FIFO` policy: the `@Override` implementation of the `add(Integer vertex)` method adds the vertex at the end of the queue data structure; the `@Override` implementation of the `delete()` methods removes and returns the least recently inserted vertex.

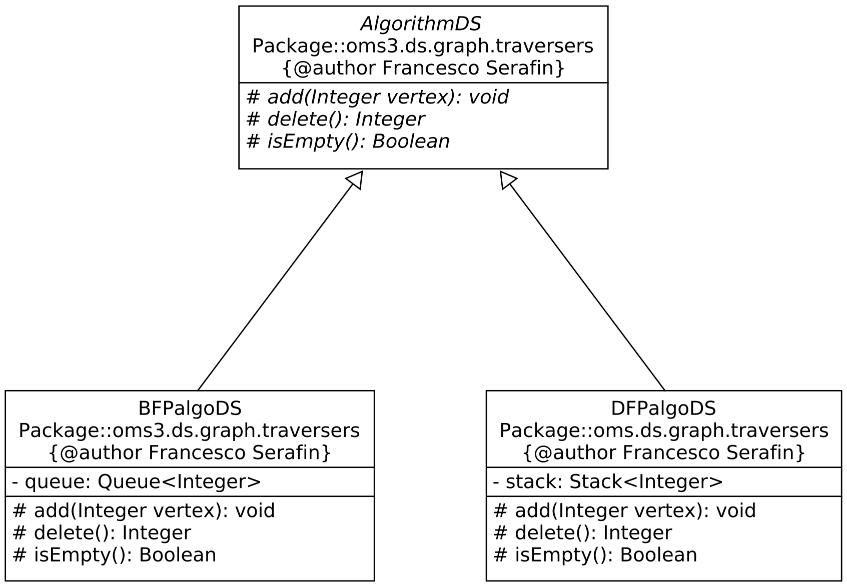

AlgorithmDS
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# add(Integer vertex): void
# delete(): Integer
# isEmpty(): Boolean
BFPalgoDS
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
- queue: Queue<Integer>
# add(Integer vertex): void
# delete(): Integer
# isEmpty(): Boolean
DFPalgoDS
Package::oms.ds.graph.traversers
{@author Francesco Serafin}
- stack: Stack<Integer>
# add(Integer vertex): void
# delete(): Integer
# isEmpty(): Boolean

Figure 79: UML of search algorithm data structures.

Listing 4.48: Breadth first path data structure implementation.

```java
/**
 *
 * @author Francesco Serafin
 */
class BFPalgoDS extends AlgorithmDS {

  Queue<Integer> queue;

  protected BFPalgoDS() {
    queue = new PriorityQueue<>();
  }

  @Override
  protected void add(Integer vertex) {
    queue.add(vertex);
  }

  @Override
  protected Integer delete() {
    return queue.remove();
  }

  @Override
  protected Boolean isEmpty() {
    return queue.isEmpty();
  }

}
```

The `DFPalgoDS` data structure is a `Stack`, which supports the LIFO policy: the `@Override` implementation of the `add(Integer vertex)` method pushes the vertex at the begging of the stack data structure; the `@Override` implementation of the `delete()` methods removes and returns (pops) the most recently inserted vertex.

Listing 4.49: Depth first path data structure implementation.

```java
/**
 *
 * @author Francesco Serafin
 */
class DFPalgoDS extends AlgorithmDS {

  Stack<Integer> stack;

  protected DFPalgoDS() {
    stack = new Stack<>();
  }

  @Override
  protected void add(Integer vertex) {
    stack.push(vertex);
  }

  @Override
  protected Integer delete() {
    return stack.pop();
  }

  @Override
  protected Boolean isEmpty() {
    return stack.isEmpty();
  }

}
```

The `DiGraph` is a directed graph. Thus, a client application might want to search for paths starting from the source node in two directions: upstream and downstream (Figure 80).

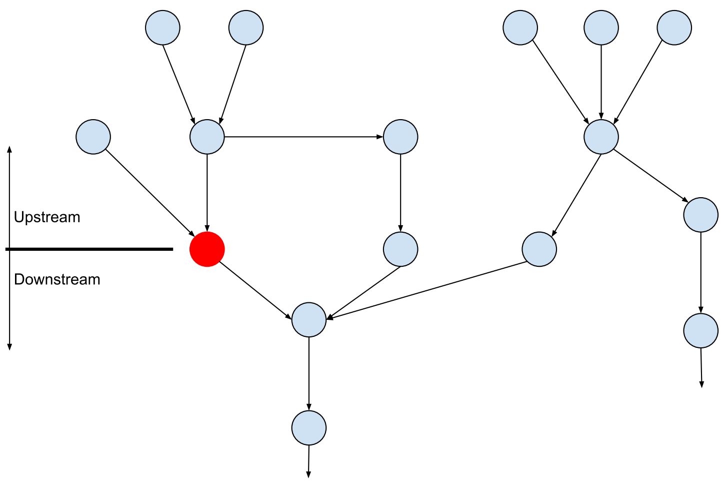


Figure 80: Concept of search upstream or downstream.

The `SearchDirection` abstract class implements *Strategy Pattern* to encapsulate the common behaviour of searching towards upstream and downstream directions. It enhances flexibility by composing its data structure with `AlgorithmDS` abstract class (Figure 81).

`SearchDirection` implements against `AlgorithmDS` API: `isDone()` returns if the `AlgorithmDS` is empty, `delete()` deletes a vertex from the data structure, while `add(Integer vertex)` adds a vertex to the data structure.

`SearchDirection` has the additional field path, which is implemented as an `ArrayDeque` of `Integer` Objects. This field stores the path from the source

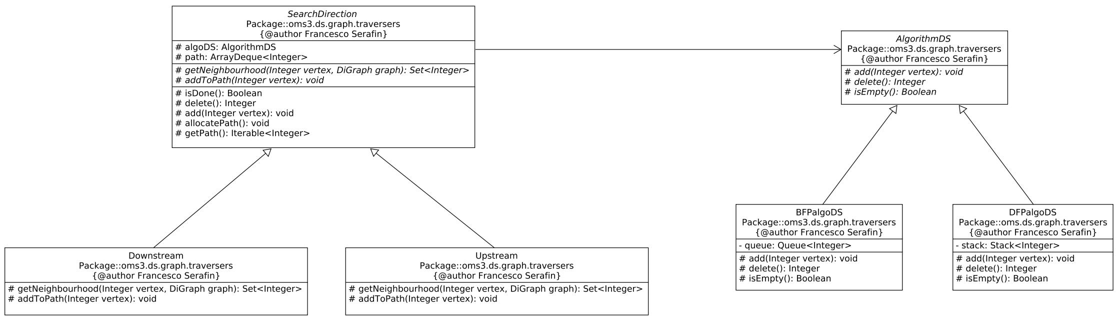

SearchDirection
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# algoDS: AlgorithmDS
# path: ArrayDeque<Integer>
# getNeighbourhood(Integer vertex, DiGraph graph): Set<Integer>
# addToPath(Integer vertex): void
# isDone(): Boolean
# delete(): Integer
# add(Integer vertex): void
# allocatePath(): void
# getPath(): Iterable<Integer>
AlgorithmDS
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# add(Integer vertex): void
# delete(): Integer
# isEmpty(): Boolean
Downstream
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# getNeighbourhood(Integer vertex, DiGraph graph): Set<Integer>
# addToPath(Integer vertex): void
Upstream
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# getNeighbourhood(Integer vertex, DiGraph graph): Set<Integer>
# addToPath(Integer vertex): void
BFPalgoDS
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
- queue: Queue<Integer>
# add(Integer vertex): void
# delete(): Integer
# isEmpty(): Boolean
DFPalgoDS
Package::oms.ds.graph.traversers
{@author Francesco Serafin}
- stack: Stack<Integer>
# add(Integer vertex): void
# delete(): Integer
# isEmpty(): Boolean

Figure 81: UML of OMS3 available modeling simulation types.

node to the end node, which results from the searching algorithm analysis. The `getPath()` method returns the path when the searching algorithm is done.

Listing 4.50: Depth first path data structure implementation.

```java
abstract class SearchDirection {

  protected AlgorithmDS algoDS;
  protected ArrayDeque<Integer> path = null;

  abstract protected Set<Integer> getNeighbourhood(Integer vertex,
                                                   DiGraph graph);

  abstract protected void addToPath(Integer vertex);

  protected Boolean isDone() {
    return algoDS.isEmpty();
  }

  protected Integer delete() {
    return algoDS.delete();
  }

  protected void add(Integer vertex) {
    algoDS.add(vertex);
  }

  protected void allocatePath() {
    if (path != null)
      throw new IllegalStateException();

    path = new ArrayDeque<>();
  }

  protected Iterable<Integer> getPath() {
    if (path == null)
      throw new IllegalStateException();

    return path;
  }

}
```

The two abstract methods `getNeighbourhood` and `addToPath` defines the actual behaviour of `SearchDirection` extended classes: `Upstream` looks for neighbours in vertex's children while `Downstream` in vertex's parents; `Ustream` adds the vertex to the end of the dequeue while `Downstream` to the front of the dequeue.

Listing 4.51: Upstream searching algorithm implementation.

```java
class Upstream extends SearchDirection {

  protected Upstream(AlgorithmDS algoDS) {
    this.algoDS = algoDS;
  }

  @Override
  protected Set<Integer> getNeighbourhood(Integer vertex,
                                          DiGraph graph) {
    return graph.getChildren(vertex);
  }

  @Override
  protected void addToPath(Integer vertex) {
    path.add(vertex);
  }

}
```

Listing 4.52: Downstrean searching algorithm implementation.

```java
class Downstream extends SearchDirection {

  protected Downstream(AlgorithmDS algoDS) {
    this.algoDS = algoDS;
  }

  @Override
  protected Set<Integer> getNeighbourhood(Integer vertex,
                                          DiGraph graph) {
    return graph.getParents(vertex);
  }

  @Override
  protected void addToPath(Integer vertex) {
    path.push(vertex);
  }

}
```

Abstract implementation of searching algorithms: Factory Method Pattern – `oms3.ds.graph.traversers.GraphSearchAlgo.java`

The implemented Strategy Pattern for both `AlgorithmDS` and `SearchDirection` in addition to composition of `SearchDirection` with `AlgorithmDS` provide for four different searching options: depth first upstream and downstream, and breadth first upstream and downstream.

The Java classes described so far still miss the actual searching algorithm implementation. In order to keep on designing a flexible and expandable digraph traversing framework, the processing algorithm has to be encapsulated into a single abstract class, and decoupled from concrete instantiations of:

A. `DiGraph` representation;

B. Searching algorithm type;

C. Searching direction.

This architectural design allows each and every object relationship within the framework to be described and managed through abstract classes (Gamma (1995)). Eventually, the choice of the algorithm to instantiate is deferred to subclasses outside the framework (Gamma (1995)).

This class creational pattern is the behaviour of a *Factory Method Pattern*: an interface rules the object creation, but actual class instantiation is deferred to subclasses (Gamma (1995)). Figure 82 illustrates the generic UML of a *Factory Method Pattern*.

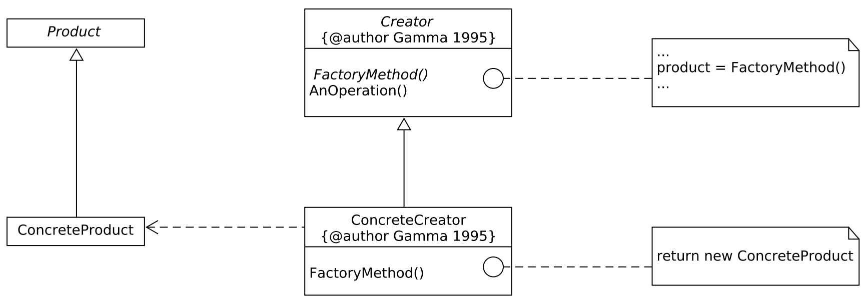


Figure 82: Generic factory method design pattern, credit Gamma (1995).

The `Creator` interface delegates the responsibility of object instantiation to the actual `ConcreteCreator` class, which manufactures the `ConcreteProduct` in the `FactoryMethod()` implementation. The `Creator` interface becomes an abstract class when it contains implementation of operational methods as well.
Figure 83 shows the UML of the pattern actually implemented.

Here, abstract class `GraphSearchAlgo.java` is the `Creator` interface. It delegates the responsibility of object instantiations to `DepthFirstPaths.java` and `BreadthFirstPaths.java`. Listing 4.53 shows the source code of `GraphSearchAlgo`.

Listing 4.53: NET3 graph search algorithm implementation.

```java
public abstract class GraphSearchAlgo {

  private Map<Integer, Integer> edgeTo;
  private Map<Integer, Boolean> marked;
  private SearchDirection searchDir;
  private Integer source;

  public void compute(String direction,
                      Integer source, DiGraph graph) {
    initialize(graph, source);
    searchDir = buildAlgo(direction);

    marked.put(source, Boolean.TRUE);
    searchDir.add(source);
    while (!searchDir.isDone()) {
      Integer vertex = searchDir.delete();
      searchDir.getNeighbourhood(vertex, graph)
               .forEach(neighbour -> {
        if (!marked.get(neighbour)) {
          edgeTo.put(neighbour, vertex);
          marked.put(neighbour, Boolean.TRUE);
          searchDir.add(neighbour);
        }
      });
    }
  }

  private void initialize(DiGraph graph, Integer source) {
    this.source = source;
    this.edgeTo = new ConcurrentHashMap<>();
    this.marked = new ConcurrentHashMap<>();
```

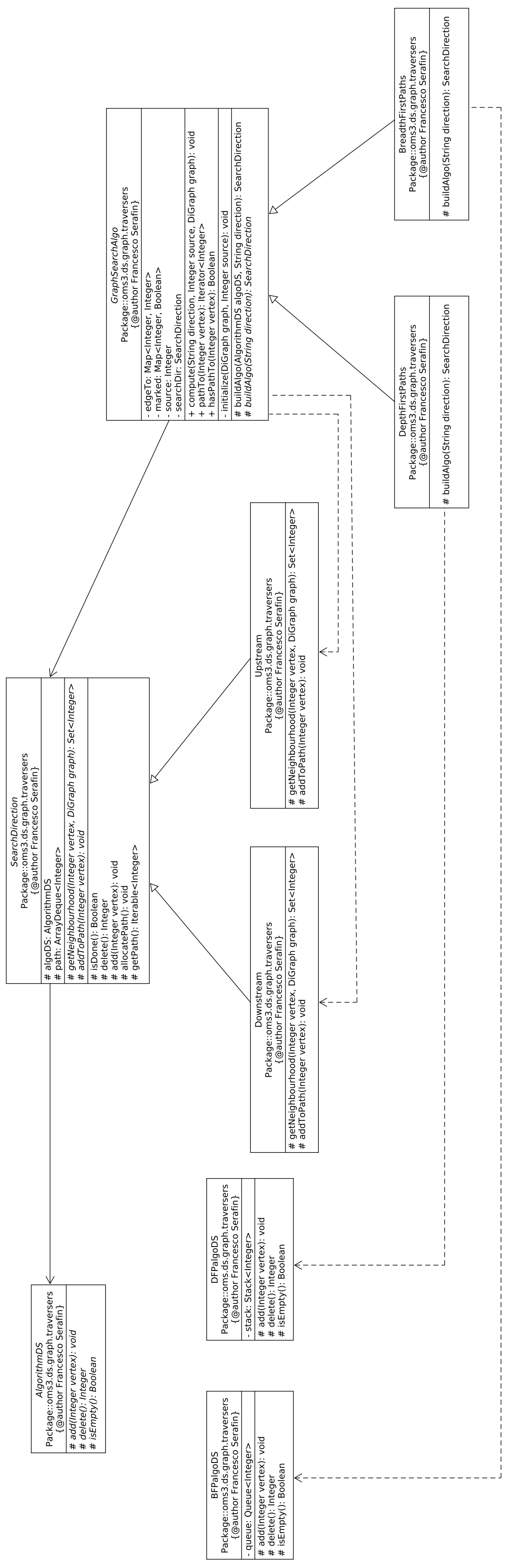

GraphSearchAlgo
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
- edgeTo: Map<Integer, Integer>
- marked: Map<Integer, Boolean>
- source: Integer
- searchDir: SearchDirection
+ compute(String direction, Integer source, DiGraph graph): void
+ pathTo(Integer vertex): Iterator<Integer>
+ hasPathTo(Integer vertex): Boolean
- initialize(DiGraph graph, Integer source): void
# buildAlgo(AlgorithmDS algoDS, String direction): SearchDirection
# buildAlgo(String direction): SearchDirection
BreadthFirstPaths
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# buildAlgo(String direction): SearchDirection
DepthFirstPaths
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# buildAlgo(String direction): SearchDirection
Upstream
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# getNeighbourhood(Integer vertex, DiGraph graph): Set<Integer>
# addToPath(Integer vertex): void
SearchDirection
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# algoDS: AlgorithmDS
# path: ArrayDeque<Integer>
# getNeighbourhood(Integer vertex, DiGraph graph): Set<Integer>
# addToPath(Integer vertex): void
# isDone(): Boolean
# delete(): Integer
# add(Integer vertex): void
# allocatePath(): void
# getPath(): Iterable<Integer>
Downstream
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# getNeighbourhood(Integer vertex, DiGraph graph): Set<Integer>
# addToPath(Integer vertex): void
AlgorithmDS
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
# add(Integer vertex): void
# delete(): Integer
# isEmpty(): Boolean
DFPalgoDS
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
- stack: Stack<Integer>
# add(Integer vertex): void
# delete(): Integer
# isEmpty(): Boolean
BFPalgoDS
Package::oms3.ds.graph.traversers
{@author Francesco Serafin}
- queue: Queue<Integer>
# add(Integer vertex): void
# delete(): Integer
# isEmpty(): Boolean

Figure 83: UML of NET3 factory method design pattern.

```
    graph.getVertecesIndeces().forEach(index -> {
      this.marked.put(index, Boolean.FALSE);
    });
  }

  public Boolean hasPathTo(Integer vertex) {
    return marked.get(vertex);
  }

  public Iterator<Integer> pathTo(Integer vertex) {
    if (!hasPathTo(vertex)) {
      return null;
    }

    searchDir.allocatePath();
    for (int i = vertex; i != source; i = edgeTo.get(i)) {
      searchDir.addToPath(i);
    }
    searchDir.addToPath(source);
    return searchDir.getPath().iterator();
  }

  protected SearchDirection buildAlgo(AlgorithmDS algoDS,
                                      String direction) {
    if (direction.equals("upstream")) {
      return new Upstream(algoDS);
    } else if (direction.equals("downstream")) {
      return new Downstream(algoDS);
    } else {
      // Add msg
      throw new UnsupportedOperationException();
    }
  }

  abstract protected SearchDirection buildAlgo(String direction);
}
```

From a *Factory Method Pattern* standpoint, the implementation of abstract `buildAlgo` method in `DepthFirstPaths.java` and `BreadthFirstPaths.java` is a simple call to **`super`**`.buildAlgo` with a new instantiation of the appropriate data structure and the search direction (see Listing 4.54 a) and b)).

Listing 4.54: NET3 encapsulation of depth first and breadth first.

```
public class DepthFirstPaths extends GraphSearchAlgo {

  @Override
  protected SearchDirection buildAlgo(String direction) {
    return super.buildAlgo(new DFPalgoDS(), direction);
  }

}

public class BreadthFirstPaths extends GraphSearchAlgo {

  @Override
  protected SearchDirection buildAlgo(String direction) {
    return super.buildAlgo(new BFPalgoDS(), direction);
  }

}
```

`GraphSearchAlgo` API doesn't expose any *Factory Method*. It exposes the method compute, which (1) initializes the required data structures, (2) instantiates the `SearchDirection`, (3) computes the actual searching from the source node;

and the method pathTo, which returns the path to a target node. Additionally, it exposes a hasPathTo method, which allows for checking if a path exists between source and target nodes.

4.4.2.2.3 Graph simulation

This section describes the integration of NET3 data structures and searching algorithms into OMS3 core.

Subparagraph *NET3 DSL* describes the interface between user and NET3 framework capabilities. Subparagraph *NET3 implicit parallelization* introduces to architectural design of NET3 implicit parallelization of independent modeling solutions run. Finally, the interconnection of modeling solutions is analyzed in subparagraph *NET3 memory management*.

NET3 DSL

In order to simplify NET3 user experience and fully integrate NET3 capabilities into OMS3 modeling framework, the available DSL has been expanded.

One main simulation file allows for setting up NET3 input entries and rules the workflow execution.

The graph simulation DSL file is expected to have a *.sim extension and adhere to the identical formal structure of a standard Sim file.

Listing 4.55 shows a typical graph simulation DSL. The initial testing of NET3 was carried out through Dr Bancheri's applications. As a result, the following DSLs are extracted from Bancheri (2017) and Bancheri et al. (2018a).

Listing 4.55: Net3 DSL sim file.

```
1  import static oms3.SimBuilder.instance as OMS3
2
3
4  OMS3.graph(path: "./data/Basento/topoBasento_106.csv",
5            simpath: "./simulation_basento/simulation_calibrazione/") {
6
7      build()
8
9      graph(traverser: "downstream.all") {
10       parameter(file: "$oms_prj/data/Basento/mixed_params.csv")
11     }
12
13 }
```

NET3 is invoked by using OMS3.graph() at Line 8 Listing 4.55, which is the identical call to OMS3.sim simulation.

As a matter of fact, the AbstractSimulation.java class is now extended by SimGraph as well, the new simulation DSL which resembles the behaviour of the standard Sim DSL, but rules the overall execution of one simulation file in each node of the graph modeling structure (Figure 84).

To maintain the analogy with a standard OMS3 modeling solution, the graph section at Line 13 Listing 4.55 is identical to the model section of an OMS3 sim file.

The path variable at Line 8 Listing 4.55 is mandatory to build the Graph model first, and DiGraph data structure secondly. That variable points to NET3 topology file, which is text file that describes the modeling solution interconnections: it lists space separated <child> <parent> relationships (child and parent are two integers), it must have at least one root (parent with index 0), and it cannot contain any loop.

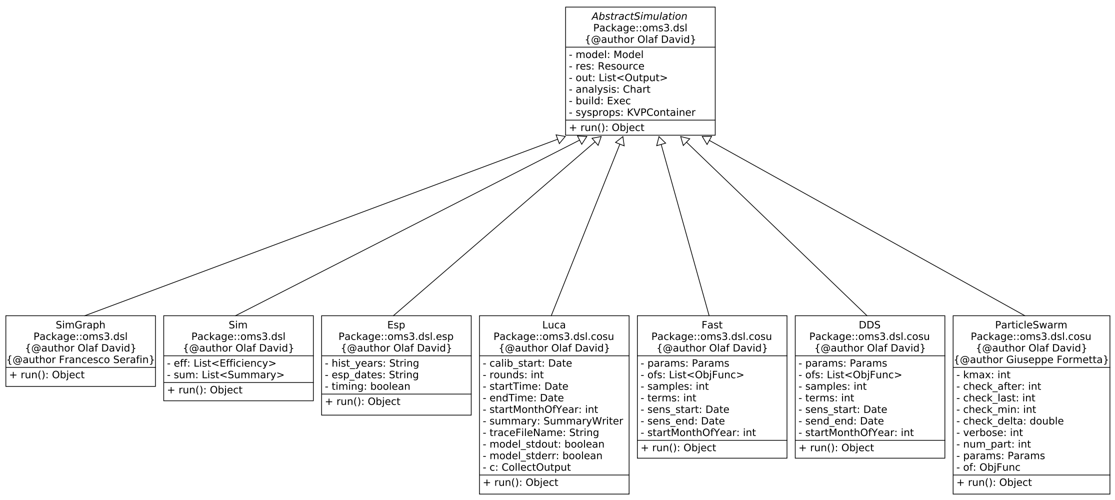


Figure 84: UML of OMS3 available modeling simulation types including NET3.

Each node index listed in the topology file corresponds to a simulation file that rules the node dedicated modeling solution such as Figure 1. That simulation file is name `<node_index>.sim`.

The traverser of the `Graph` model (Line 13 Listing 4.55) is the NET3 parameter that rules `DiGraph` data structure traversing: it allows for analyzing the entire network topology or extracting and analyzing a sub-branch only.

The traverser parameter adheres to the following formal structure: `<searchDirection>.<algorithm>`. Listing 4.55 illustrates the most common application: the entire network topology is analyzed by following natural connections (downstream) between each node of the structure (all).

Beta functionalities (still under development and not fully tested yet) allow for parsing one subbranch of the structure only. The traverser might be coupled to a foreach parameter to adhere to the following formal structure:

Listing 4.56: Formal structure of NET3 search direction DSL.

```
graph(traverser: "<seachDirection>.<algorithm>",
      foreach: "from_node -> to_node") {
      [...]
}
```

Here, `searchDirection` can be `downstream` or `upstream`, algorithm can be `breadthfirst` or `depthfirst`. With respect to foreach parameter, `from_node` and `to_node` are two indices in the network topology. If the two nodes are connected, the subbranch is extracted and analyzed, otherwise an error message is thrown.
A NET3 subbranch analysis example is shown in Listing 4.57.

Listing 4.57: Usage example of NET3 search direction DSL.

```
graph(traverser: "downstream.breadthfirst",
      foreach: "10 -> 4") {
      [...]
}
```

Once again, this functionality has not been fully tested yet and is currently under development. However, it is fundamental architectural aspect and initial step for furtherly designing automated parallel multi-site calibrations.

The parameter sub element in Listing 4.55 resembles the behaviour of parameter sub element in subparagraph *Parameters (parameter)*. User can specify a file as well as a list of parameters which are then used to overwrite parameters defined in every modeling simulation which runs in each node of the modeling structure.

When the Groovy runtime parses NET3 sim file, a `SimGraph` simulation is instantiated. Successively, a `Graph` model is instantiated, which in turn instantiates and manages the `DiGraph` data structure.

NET3 implicit parallelization

When the entire graph simulation has been built and instantiated by Groovy runtime, the run method in `SimGraph` class is invoked (Listing 4.58).

Listing 4.58: NET3 run method.

```
@Override
public Object run() throws Exception {
  super.run();

  [...]

  if (setpath == null) {
    setpath = graph.newModelComponent();
  }

  int availProc = (numCores != null) ?
          numCores : Runtime.getRuntime().availableProcessors();

  ExecutorService executor =
          Executors.newFixedThreadPool(availProc);

  RunSimulations sim =
          new RunSimulations(setpath, executor, availProc);
  sim.run();
  executor.shutdown();

  return comp;
}
```

Here, the `Graph` model returns the `ConcurrentLinkedDeque<Integer>` object, which contains the indices selected by the searching algorithm during the Groovy build runtime process. In the most generic case, the dequeue with every single node in the graph modeling structure is returned.

Afterwords, a new `ExecutorService` with the number of available processors is instantiated. The private class `RunSimulations` manages the entire graph model run.

Depending on the number of available processors, the number of concurrent simulation threads are submitted. Each thread goes through the `ConcurrentLinkedDeque` looking for a node of the network topology ready to run.

This algorithm works on a *Observer Pattern*-like design. The logic behind this architectural design is following described (each available processing thread performs this algorithm):

1. the index of a node of the network topology to be processed is extracted from the `ConcurrentLinkedDeque`;

2. the thread holding the extracted index checks its availability to run (all its upstream nodes have already run);

3. if the node is not ready to run the next index in the `ConcurrentLinkedDeque` is extracted and the algorithm starts from point 1 again;

4. if the node is ready to run, input requirements get satisfied (see Subparagraph *NET3 memory management*) and the modeling solution named with the node index is executed;

5. once the modeling solution execution is over, its outputs are pushed to a memory buffer (see Subparagraph *NET3 memory management*);

6. finally, node's parents get notified that simulation of this children is done.

This logic resembles the one-to-many dependency behaviour of an *Observer pattern*. In this pattern there are two actors involved: a subject (or publisher) and an observer (or subscriber). The observer is interested in getting notified when the subject changes its state. Consequently, it subscribes to receive notifications. Any number of observers can subscribe to receive information from one subject, and an observer can subscribe to any number of subjects. Each subject doesn't know anything about who subscribed to get its notifications, it simply publishes the change of its state. The observer queries the subject state to update the synchronization (Gamma (1995)).

NET3 resembles this behaviour since parents subscribe to receive notification of their children status. When the `DiGraph` is instantiated by the Groovy runtime, the initialize method (Figure 81) creates an `HashMap<Integer, Boolean>` for each node in the vertices data structure.

Listing 4.59: NET3 observer initialization.

```java
public void initialize() {
  vertices.keySet().forEach((vertex) -> {
    Family fam = edges.get(vertex);
    HashMap<Integer, Boolean> hm = (HashMap) vertices.get(vertex);
    if (fam.childrenNumber() != 0) {
      for (Integer child : fam.children) {
        hm.put(child, Boolean.FALSE);
      }
    } else {
      hm.put(0, Boolean.TRUE);
    }
    vertices.replace(vertex, hm);
  });
}
```

This `HashMap` is initialized with value `Boolean.FALSE` for each key, which is the child index. `Boolean.FALSE` means that the simulation of that child is not done yet. When the simulation of the child is over, the `Graph` model notifies the parent that child computation is done by calling the parentNotice method.

This notifies each child's parent that child state changed and replaces `Boolean.FALSE` with `Boolean.TRUE`.

Listing 4.60: NET3 observer notification.

```java
public void parentNotice(Integer vertex) {
  edges.get(vertex).parents.forEach(parent -> {
    HashMap<Integer, Boolean> hm =
          (HashMap<Integer, Boolean>) vertices.get(parent);
    hm.replace(vertex, Boolean.TRUE);
    vertices.replace(parent, hm);
  });
}
```

The `readyForSim` method is queried by each thread with the node index each thread is holding (step 2 of the `RunSimulation` algorithm). This method returns true only if node's children have value `Boolean.TRUE` in the hashmap corresponding.

Listing 4.61: NET3 ready for simulation check.

```java
public boolean readyForSim(Integer vertex) {
  return !((HashMap<Integer, Boolean>)
            vertices.get(vertex)).containsValue(false);
}
```

NET3 memory management

Communication between interconnected nodes happens through a thread safe memory buffer. This buffer is a common memory space where each node of the graph modeling solution pushes its output to when the simulation run is over.

Before starting a model run, each node satisfies input requirements by pulling input data from the memory buffer. The previously described observer-like modeling run makes sure that a node accesses to pull methods when its children have pushed their output already.

Node inputs/outputs have to be identified from within the proper model component. In order to keep the lightweight invasiveness of OMS3, the annotation functionalities have been expended by adding two new annotations: `@InNode` (incoming variable) and `@OutNode`(outcoming variable).

Listing 4.62: Example of a POJO class turned into OMS/NET3-compliant component by accommodating OMS3 and NET3 annotations.

```java
package example;

import oms3.annotations.*;

public class CylinderVolume {

  @InNode
  public double radius;

  @In
  public double height;

  @OutNode
  public double volume;

  @Execute
  public void compute() {
    volume = circleArea(radius) * height;
    System.out.println("The volume is: " + volume);
  }

  private double circleArea(double radius) {
    return Math.pow(radius, 2) * Math.PI;
  }

}
```

OMS3 requires that components connections are specified in the `*.sim` file (see Subparagraph *Component connections (connect)*). A similar policy is required to connect modeling solutions. The model element of a sim file has been enriched with two additional sub elements: `inFluxes` and `outFluxes`. Listing 4.63 shows an usage example.

Listing 4.63: inFluxes/outFluxes DSL usage example.

```
/*
 * Hello 'world' example.
 * A component printing a greeting.
 */
import static oms3.SimBuilder.instance as OMS3

OMS3.sim() {

  // build()

  model(while: "c.goon") {

    components {
      "in1" "ex0.In1"
      "c" "ex0.Component"
      "out1" "ex0.Out1"
    }

    parameter {
      "c.val" 14.0
      "c.other_val" 1.0
    }

    inFluxes {
      "15.outval1" "in1.inval1"
      "15.out_other_val1" "in1.in_other_val1"
    }

    outFluxes {
      "out1.outval1" "
      "out1.out_other_val1" "
    }

    connect {
      "in1.outval" "c.inval"
      "c.outval" "out1.inval"
      "in1.out_other_val" "c.in_other_val"
      "c.out_other_val" "out1.in_other_val"
    }
  }
}
```

`inFluxes` sub element adheres to the following formal structure:

Listing 4.64: inFluxes formal structure.

```
  inFluxes {
    <child_node_A>.<variable> <cID_1>.<variable>
    <child_node_B>.<variable> <cID_2>.<variable>
}
```

The parent node needs to specify which child to pull specific information from. `outFluxes` sub element adheres to the following formal structure:

Listing 4.65: outFluxes formal structure.

```
  outFluxes {
    <cID_1>.<variable> ""
    <cID_2>.<variable> ""
  }
```

Here, the output variables have to be simply listed and associated to an empty String.

4.5 CASE STUDIES

As previously stated, Dr. Bancheri continuously tested every new NET3 improvement and asked for implementation of some functionalities like parameter overwriting and first requirements for multi-site calibration development. As a result, 4.5.1 is the first application following introduced (Bancheri (2017); Bancheri et al. (2017a, 2018a)). The author of this dissertation doesn't take any credit with respect to 4.5.1 application development. This application is described in this dissertation since it leverages NET3 functionalities.

Additionally, two further applications have been developed to exercise and demonstrate NET3 capabilities applied to generic complex networks problems.

JSWMM is a Java-based redesigned version of Storm Water Management Model (SWMM), urban hydrology model part of the GEOframe system. The innovative architectural design of JSWMM utilizes the benefits of OMS3-NET3 capabilities to (1) split monolithic code base into conceptual/physical processes and encapsulate them into modeling components, and (2) avoid code duplication by reusing identical modeling solutions to model run off and routing in each node of a complex sewer network. This application is Mr. Dalla Torres's master's project (Dalla Torre et al. (2018)). The author of this dissertation directly supervised Mr. Dalla Torre along architectural design and software development of his master's project. Mr. Dalla Torre holds the authorship of software engineering design and source code of JSWMM.

System of systems of models (SSoM) is a multi-languages multi-models application developed for the Framework for Integrating the Complexity of Uncertain Systems (FICUS) (Burkhalter et al. (2018); Ehlschlaeger et al. (2014b,a, 2018b,c,a); Westervelt et al. (2017)). This application is built upon NET3 capabilities of encapsulating independent modeling solutions in separated nodes, and connecting interoperable nodes based off of input/output relationships. Here, modeling solutions run different models, which are also implemented in different programming languages (Serafin et al. (2018c)). The author of this dissertation doesn't take any credit with respect to code base of models included in the SSoM application (Transims and HISA) and set up of their input files. The author of this dissertation designed the graph modeling solution, developed required NET3 capabilities, developed OMS3 R and Python bindings (Serafin et al. (2018c)), developed OMS-compliant Python wrappers to run Transims components, and implemented highly parallelized OMS-compliant R plotting components.

These two additional applications are part of this dissertation because each one of them required development of specific NET3 features, which are following described.

Each subsection is organized with an *Application* paragraph to introduce to the application itself, and a *NET3 additional features* paragraph to summarize dedicated NET3 features developed.

4.5.1 GEOframe: Monitoring hydrological extremes

4.5.1.1 *Application*

This application is a Decision Support System (Decision Support System (DSS)) developed with the final goal of monitoring and forecasting hydrological extreme events in Basilicata region, Italy (Grasso et al. (2017); Bancheri et al. (2018a)).

The DSS has been implemented on front/back ends software engineering design. The separation of concerns principle has been elevated to split the representation layer (front end: WebGIS Figure 85) from the computational modeling layer (back end: GEOframe)(Bancheri et al. (2018a)).

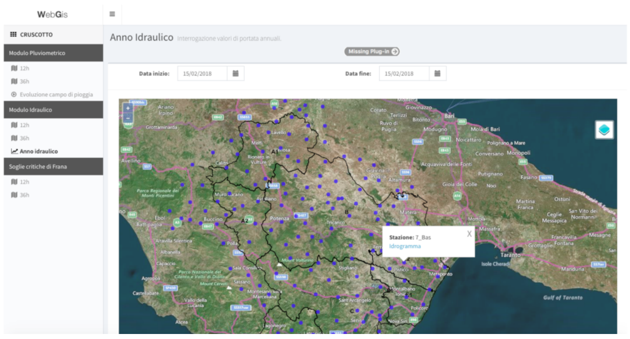


Figure 85: Webgis front-end, credit Bancheri et al. (2018a).

Here, the back end is a graph modeling solution developed on GEOframe semi-distributed physically based hydrological system and OMS3-NET3 software framework.

To simulate hydrological and hydraulic variables in near-real time, the entire Basilicata region was divided into about 160 interconnected HRUs (Bancheri et al. (2018a)) (Figure 86).

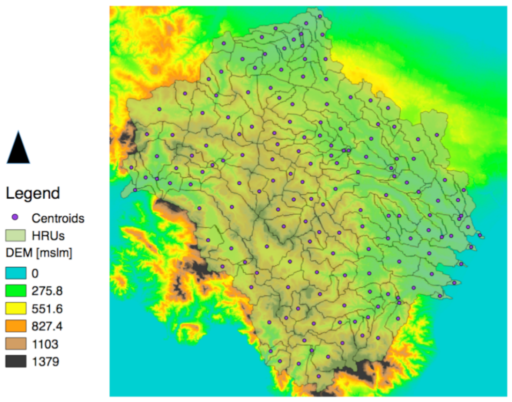


Figure 86: HRUs, credit Bancheri et al. (2018a).

This infrastructure results in a complex network that perfectly fits NET3 modeling capabilities. Figure 1 at the beginning of this dissertation shows the runoff modeling solution computed in each node of graph modeling structure.

4.5.1.2 *NET3 additional features*

This application, along with Bancheri (2017), is the main unit test the development of NET3 has been based on.

In addition to main features (e.g. implicit parallelization, integration of NET3 with LUCA calibration), two supplementary capabilities have been specifically designed for this application:

1. parameterization per HRU;
2. multi-site calibration.

4.5.1.2.1 Parameterization per HRU

This feature allows for setting up a dedicated set of parameters per HRU. The parameters can be easily tuned from within modeling solution of each node of the GMS. Additionally, the main graph simulation file allows for:

1. defining a set of parameters that are potentially common to several modeling solution in the parameter sub element of the graph element (Section NET3 DSL Listing 4.66). The additional flag element list which node to overwrite parameters to;
2. defining an extra `paramFile` element that lists space separated node number and parameter files to read parameters from.

Listing 4.66: NET3 flags and paramfiles extensions.

```
import static oms3.SimBuilder.instance as OMS3

OMS3.graph(path: "$oms_prj/.../topoBradano_SanGiuliano.csv",
           simpath: "./simulation_Bradano/sim[...]e_SanGiuliano/") {
    build()

    graph(traverser: "downstream.all") {
          parameter(file: "$oms_prj/data/Bradano/mixed_params_SG.csv")

    }

    flags {
      "1" "{overwrite}"
      "2" "{overwrite}"
      "3" "{overwrite}"
      [...]
      "35" "{overwrite}"
      "36" "{overwrite}"
      "37" "{overwrite}"
    }

    paramfiles {
      "1" "$oms_prj/data/Bradano/mixed_params_SG.csv"
      "28" "$oms_prj/data/Bradano/mixed_params_SG.csv"
      "37" "$oms_prj/data/Bradano/mixed_params_SG.csv"
    }

}
```

4.5.1.2.2 Multi-site calibration

This feature is of fundamental importance when it comes to calibrating modeling parameters per group of HRUs against internal monitoring sites within the main catchment.

Figure 87 facilitates the description of this functionality. Two monitoring site are available in this catchment of Basilicata region: Ponte La Marmora and Agri SS106. In order to make use of important available data, a first calibration procedure is

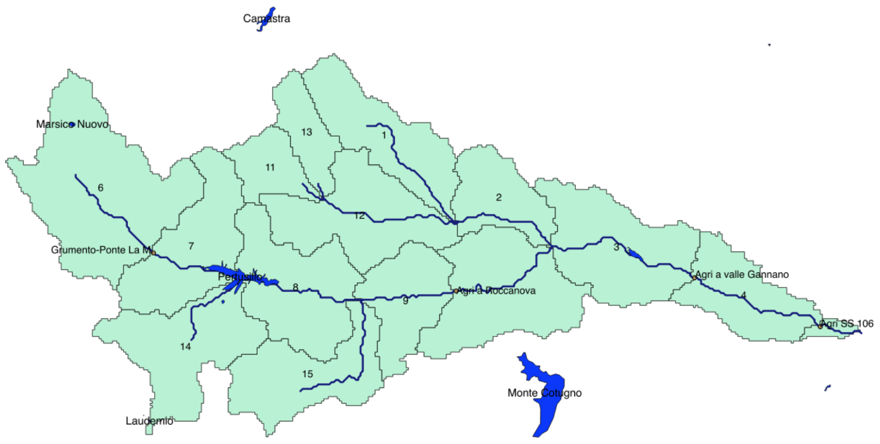


Figure 87: Multi-site calibration, credit Bancheri et al. (2018a).

set up on HRU 6. Then, calibrated parameters remain constant in HRU 6 during calibration procedure against monitoring site in Agri SS106.

NET3 allows for defining which node of the graph modeling simulation to calibrate from the element flags.

Listing 4.67: NET3 flags calibrate extension.

```
import static oms3.SimBuilder.instance as OMS3

 /*
  * Luca calibration.
  */
 OMS3.luca(name: "EFC-luca_Agri_Ponte_70",
           path: "$oms_prj/data/Agri/topoAgri_Ponte.csv",
           simpath: "./simulation_Agri/sim_calib_Ponte/") {

     graph(traverser: "downstream.all"){
       parameter(file: "$oms_prj/data/Agri/mixed_params_Ponte.csv")
     }
     flags {
       "1" "{calibrate}"
       "2" "{calibrate}"
       "3" "{calibrate}"
       "4" "{calibrate}"
       "5" "{calibrate}" // 6 is missing, won't be calibrated
       "7" "{calibrate}"
       [...]
     }

     run_start          "2013-12-15"
     calibration_start  "2013-12-15"
     run_end            "2014-12-15"
     rounds              2

     // step definitions
     step {
       parameter {

         [...]
         alfa_l      (lower:0.3, upper:0.9,calib_strategy:MEAN)

         kc_canopy_out  (lower:0.1, upper:2,calib_strategy:MEAN)
         [...]
       }

       objfunc(method:KGE, timestep:RAW,invalidDataValue:-9999) {
         sim(file:"$oms_prj/output/Agri/Calib/Idrogramma_6_Agr.csv",
             table:"table", column:"value_6")
         obs(file:"$oms_prj/data/Agri/Agri_PonteLaMarmora.csv",
             table:"table", column:"value_6")
       }
       max_exec 200
     }

}
```

Figure 88 and Figure 89 show calibration results at Ponte La Marmora and Agri SS106.

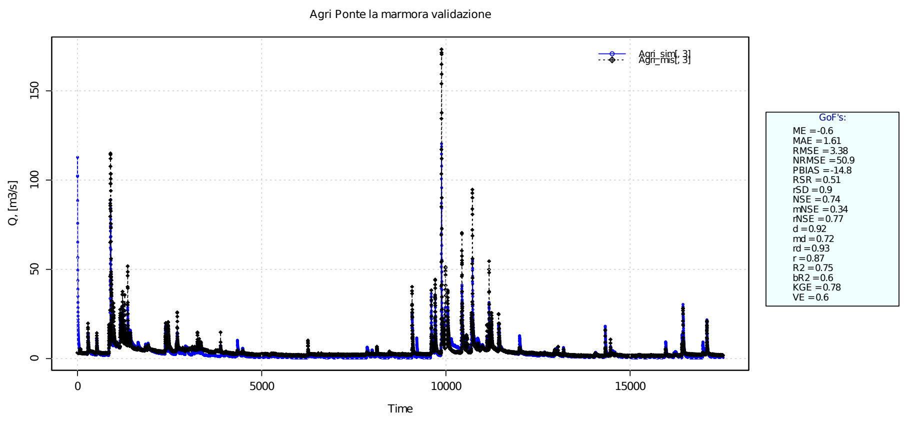


Figure 88: GEOframe validation at Ponte La Marmora, credit Bancheri et al. (2018a)

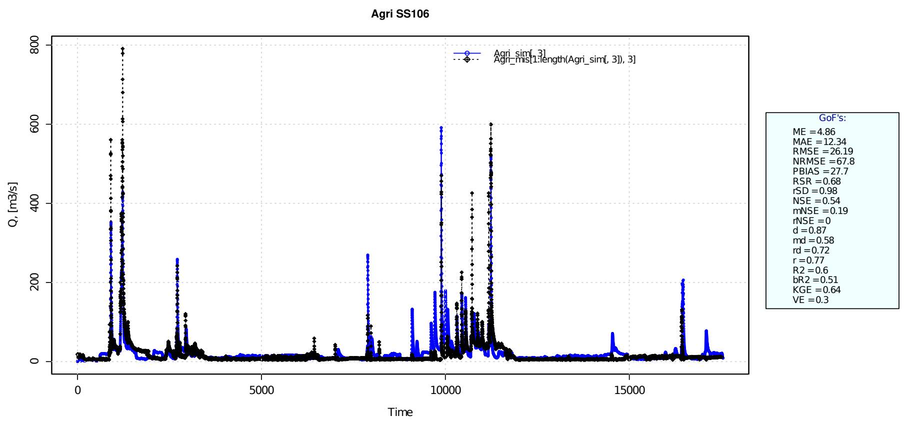


Figure 89: GEOframe validation at Agri SS106, credit Bancheri et al. (2018a)

4.5.2 GEOframe: JSWMM

4.5.2.1 *Application*

This application results from a Java-based software architectural redesign of SWMM (Storm Water Management Model), in addition to major computational modules reimplementation.

SWMM is a computational model for estimating quantity and quality of urban runoff. Its application ranges from modeling of single event to long-term simulations. It is broadly used especially for stormwater and sanitary sewer design, analysis of pollutant transport, treatment strategies of point and nonpoint sources, and urban planning (Gironás et al. (2010)).

Here, the main goals of JSWMM exercise are to: (1) redesign runoff and routing original SWMM modules as component-based OMS-compliant software, thus creating a flexible and expandable infrastructure for accommodating future developments; (2) add a design component to original SWMM workflow for actually designing the storm sewer network and not just verifying it (Dalla Torre et al. (2018); Dalla Torre (2019)). This goals are achieved without impacting on user experience, namely original SWMM input/output file format don't change.

JSWMM runoff, pipe design, and routing computational methods are encapsulated into independent components.

Instead of computing runoff, routing, and pipe design in a single modeling component per drainage area (standard modeling approach), runoff and routing/design components for each drainage area are bundled into separated nodes of the network topology. This approach allows for leveraging high computational scalability of NET3 implicit parallelization (Figure 90) (Dalla Torre et al. (2018); Dalla Torre (2019)).

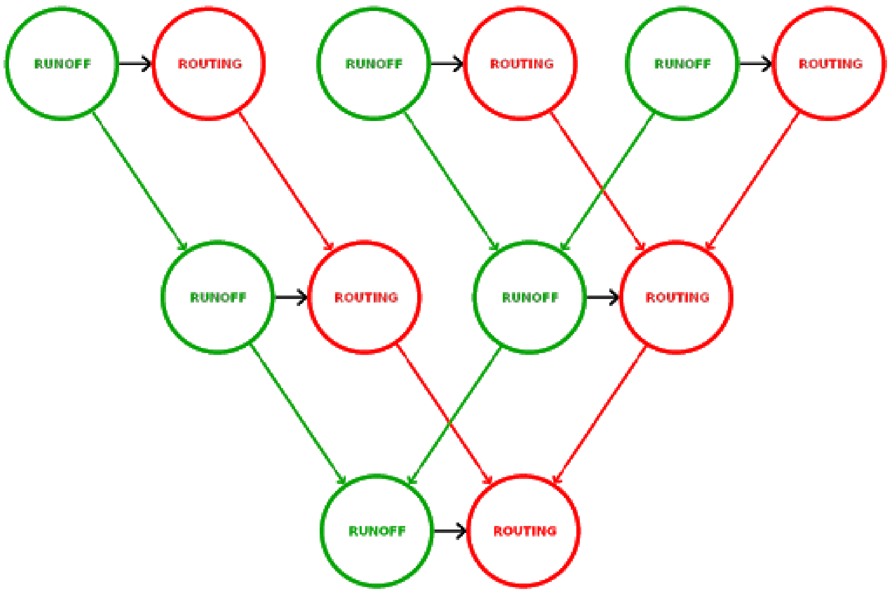


Figure 90: NET3-JSWMM component granularity, credit Dalla Torre et al. (2018); Dalla Torre (2019).

JSWMM has been tested and validated on Fossolo Network sample dataset. Figure 91 illustrates JSWMM results compared to original SWMM run out of a drainage area and overall urban network outlet.

4.5.2.2 *NET3 additional features*

Two supplementary NET3 features have been specifically designed to accommodate modeling requirements from this application:

1. allowing runoff, pipe design, and routing components to access a single data structure to resemble original SWMM workflow;
2. providing pipe design component with access to node indices of its upstream subnetwork for adjusting the depth of each upstream pipe.

These features are currently not flexibly implemented and require additional architectural design.

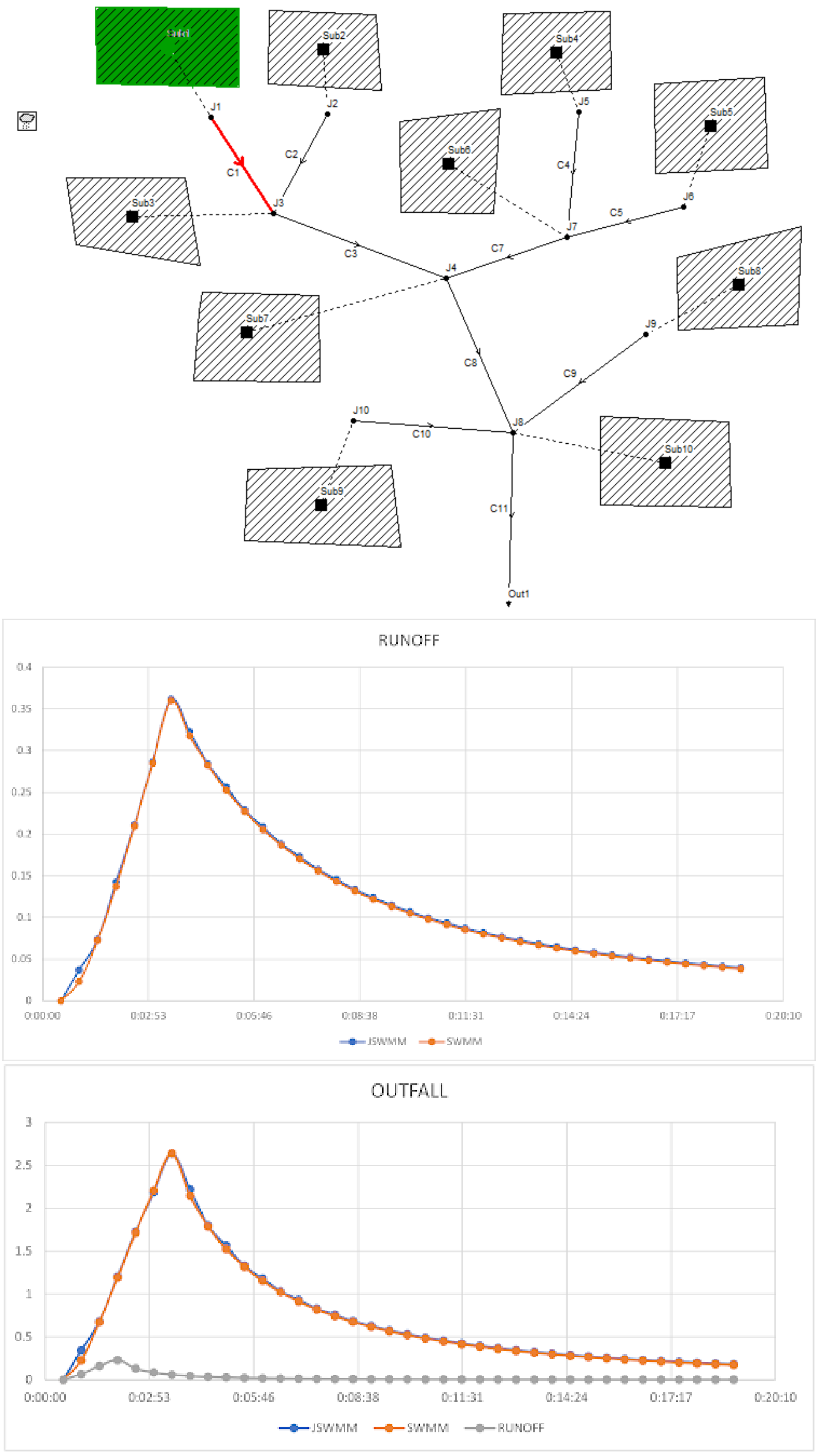


Figure 91: JSWMM modeling results, Dalla Torre et al. (2018); Dalla Torre (2019)

4.5.2.2.1 Access to common data structure

Listing 4.68 shows NET3 DSL that enables the main simulation file to push an empty data structure to the memory buffer before starting the computation on the urban network.

Listing 4.68: NET3-JSWMM access to common data structure, credit Dalla Torre et al. (2018); Dalla Torre (2019).

```
import static oms3.SimBuilder.instance as OMS3
import org.altervista.growworkinghard.jswmm.dataStructure.SWMMobject

OMS3.graph(path: "./data/topo_small.csv",
           simpath: "./simulation_test/") {

    resource "$oms_prj/lib"

    build()

    graph(traverser: "downstream.all") {

    }

    model() {

      components {
        "c" "ex0.OutT"
      }

      outFluxes {
        "c.datastructure" ""
      }
    }

}
```

Consequently, every component from each node of the graph data structure can access this common memory location and push/pull data resembling SWMM original behaviour.

4.5.2.2.2 Access to upstream sub-branch

NET3 provides a hidden variable to JSWMM pipe design component with a list of indices of upstream nodes. This allows JSWMM to adjust the depth of upstream pipes when the current pipe has been designed.

4.5.3 FICUS: System of systems of models

4.5.3.1 *Application*

This application allows for analyzing people's access to facilities such as water, fuel, etc. before and after amenity disruptions (Burkhalter et al. (2018); Ehlschlaeger et al. (2018c,a); Lu et al. (2018); Westervelt et al. (2017); Xie et al. (2019)).

Here, NET3 has been utilized as a result of the need for interconnecting different models, Transims (Smith et al. (1995)) and HISA, with different iterative loops (see node 9 and node 11 in Figure 92). NET3 allows for encapsulating completely different modeling solutions in different nodes of the modeling structure, and interconnecting them regardless.

Additionally, this application exercises OMS3-NET3 multi language interoperability by enabling seamless communication between Java, Python, and R OMS-compliant components (Serafin et al. (2018c)).

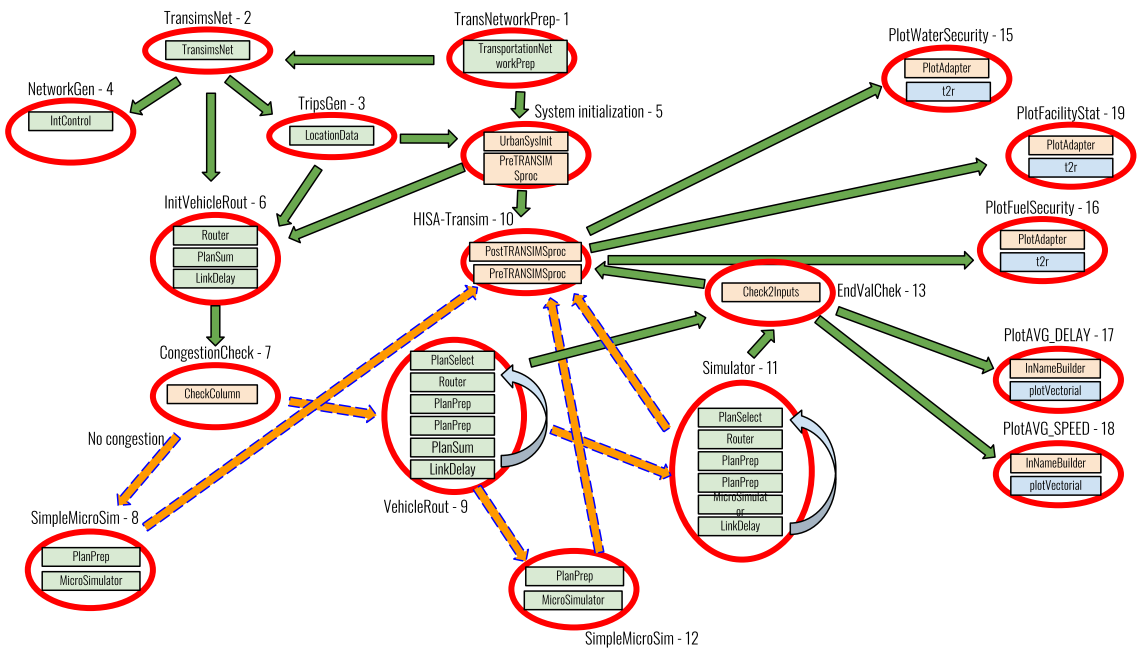


Figure 92: FICUS SSoM conceptual design.

Figure 92 illustrates an SSoM modeling solution: each node of the graph modeling structure encapsulates different types of modeling solutions. Here, orange rectangles indicates Java components (e.g. node 10), green rectangles represents Python components (e.g. node 6), and light blue rectangles indicates R components (e.g. node 15).

Figure 93 shows two screenshots of the FICUS-UI for visualization and analysis of uncertainty quantified geographically spatialized raster/vectorial data (credit Olaf David and David Patterson).

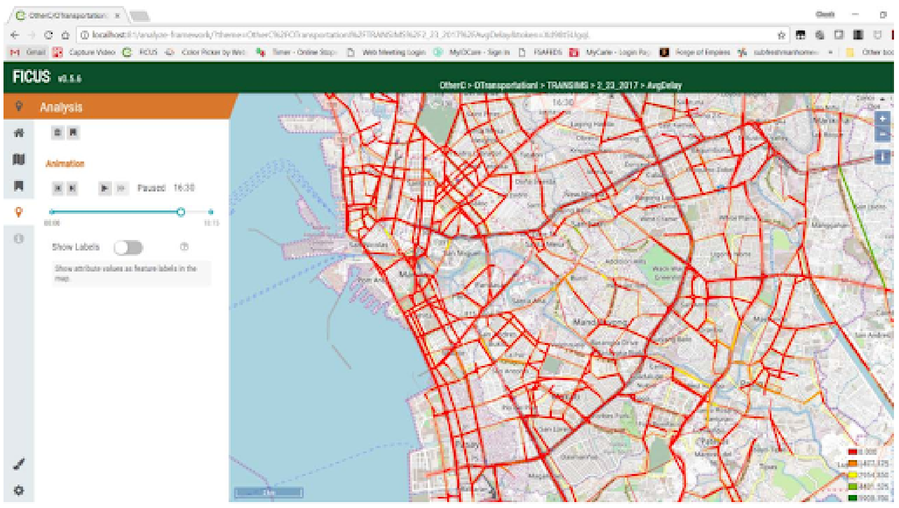


Figure 93: SSoM results displayed from the FICUS-UI.

4.5.3.2 *NET3 additional features*

Since both Transims and HISA models implement stochastic algorithms, the identical complex modeling solution in Figure 92 has to run several times to allow for generating different realizations out of the same scenario.

NET3 provides the graph of graphs feature where each node of the graph modeling structure is enabled to run an inner graph modeling structure.

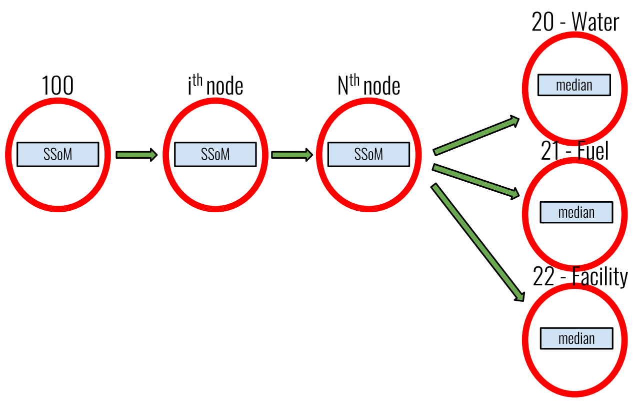


Figure 94: NET3 Graph of Graphs conceptual design.

Figure 94 illustrates the graph of graphs modeling solution. Here, node 100, node *ith*, and node *Nth* run one SSoM (Figure 92). Eventually, uncertainty quantification of water, fuel, and facility access are evaluated (node 20, 21, and 22 Figure 94).

Nodes in the graph of graphs modeling solution have to run sequentially because of design constraints in the current graph memory management.

4.6 SUMMARY

This chapter introduces to design and implementation of NET3 graph modeling structure approach.

Literature review illustrates previous applications of graph data structure in environmental modeling and identifies lack of flexibility and portability across different operating systems in addition to programming proficiency requirements for fully taking advantage of developed platforms.

Research questions identify milestones that drive analysis, design and implementation of NET3 (graph modeling structure).

Research design and methods describe methodological and technical approaches utilized to achieve flexible implementation of complex network like modeling solutions. Introduction to graph theory and its related algorithm implementation are proposed, as well as description of environmental modeling frameworks and the concept of implicit parallelism. Since EMFs is the hosting platform for fostering NET3 implementation, detailed description and analysis of OMS3 is provided. Finally, NET3 design and implementation are thoroughly examined.

Three applications exercise NET3 features. The hydrological model GEOframe is expanded and utilized as back-end for a DSS in Basilicata region, and additionally illustrates the river network - graph structure analogy. A NET3 based redesign of SWMM (JSWMM) demonstrates how a finely tuned model component granularity allows for leveraging higher level of implicit parallelization. System of Systems model exercise NET3 flexibility in a complex urban modeling environment.
Next chapter describes conclusions and future developments for FeNS and NET3.

5 CONCLUSION

Contents

Following the structure of this dissertation, conclusion and future development for Framework-enabled NEAT based Surrogate modeling (FeNS) and NET3 are separately described.

5.1 FENS: CONCLUSION AND FUTURE DEVELOPMENT

A long-known discrepancy managing physical based models exists to fully comprehend, correctly parameterize, performantly execute, and flexibly deploy them in research and service delivery environments. Research organizations fund the development of the models, but do not fund their integration into service delivery organization systems and workflows. Service delivery organizations fund integrating models into their systems and workflows, but usually find this a very difficult, inefficient, and time-consuming task, often failing because the research model has become unwieldy and too burdensome to use. For widespread frequent use, service delivery organizations need the model to compute results quickly with limited set-up, reduced data entry, taking advantage of existing organization-wide data resources.

To bridge this gap, this dissertation aims to address and alleviate research model application complexity with respect to data and parameter setup, runtime requirements, and proper model infrastructure setup. This dissertation proposes a data driven based surrogate modeling approach aiming to capture the intrinsic knowledge of a physical model into an ensemble system of artificial neural networks. This methodology accommodates application needs to get quick and "accurate enough" model results with limited input entries and limited a-priori knowledge of internal processes involved in conceptual/physical models. Here, the data-driven methodology was used as an inexact emulator of any deterministic computer simulation model.

FeNS system enables modeling framework to interact with machine learning libraries to emerge model surrogates any modelling solution. This facilitates the transitioning of mathematical models from research to field by automating the process of generation of ensemble of surrogate models. CSIP/OMS was extended and utilized to harvest data and derive the surrogate model at the modeling framework level.

FeNS demonstrates an opportunity for service delivery organizations to consolidate and streamline model delivery and application.

Recent advancements in machine learning techniques have proven to be suited for creating surrogate models of conceptual/physical models: surrogate models are capable of accurately emulate original research model behaviours by using relevant input entries only. Preliminary design of experiments show Nash-Sutcliffe accuracy above 0.95.

Surrogote models are homogeneous in their implementation and use, whereas conceptual/physical models are heterogenous in development, setup, and deployment. This provides service delivery organizations with a simplified access to mathematical models knowledge without requiring for dedicated modeling expertise and complex IT deployment infrastructure setup and management.

Surrogame models allow for more consistent deployment on server, desktop, and mobile while being platform and operating system independent, potentially run on-the-field with no internet coverage, and can be subject to dataset-like lifecycle.

Ensemble of surrogate models improves estimate accuracy by training several artifical neural networks on slightly different datasets, while uncertainty quantification of ensemble of surrogate models runs smooths the overall eSM behavioural emulation capability.

FeNS combines presented methodologies and approaches to allow for automatically emerging surrogate models from any modelling solution in a controlled environment, utilizing maximal computation resources for training, and delivering and applying ensemble of surrogate models within consultancy organizations requiring for minimal computational resources. Additionally, surrogate models necessitate of basic or no IT infrastructure management and provide uncertainty quantified answer with minimal response times. FeNS-R2 eSM with 10 artificial neural networks is about 80 KB in size and provides erosion rate estimate in around 100 ms. Opposingly, CSIP-R2 web service system is about 300 GB in size (database dependencies included) and returns erosion rate in around 10 s.

FeNS methodology and design is at early stage of development. Consequently, this dissertation identifies steps required to strengthen FeNS pipeline and facilitate its usage.

Design of Experiments demonstrate necessity of dataset clustering to fasten eSM generation and divide original modeling emulation responsibility. Currently, FeNS management of clustered data and eSMs is delegated to user manual set up. On contrary, this process has to be automated by designing an additional FeNS-service which takes care of creating required clustered data MongoDB collections.

Furthermore, discovering clusters in dataset resulting from determinist model behaviour is a trial-and-error process. Two different approaches will be investigated to attempt the automation of this delicate and important process:

A. Stanley and Miikkulainen (2002) states that continuous state spaces facilitate NEAT learning and elevate its effectiveness. Discontinuity curves from scattered data can be detected and recovered (Bozzini and Rossini (2013)). As a result, proper cluster on data discontinuities might elevate NEAT features.

B. DoE 3 has proven FeNS capability of emulating RUSLE2 behaviour on the few high erosion data available by developing a dedicated eSM cluster. The analysis of the distribution of original model responses might allows for identifying areas with scarce data. As a result, dedicated clusters might be able to improve overall eSMs accuracy by learning model behaviour off of rare/scarce data.

Finally, different gating strategy has to be tested since only one basic approach has been exercised in DoE 3.

Razavi et al. (2012a) underlines model conformability issue of ANN-based SM. Two different approaches will be investigated to automatically avoid this relevant ANN-based SM methodology issue:

A. Integrate Halton Sequence generator as part of FeNS pipeline to automatically harvest homogeneously distributed original model responses.

B. Implement Bayesian Regularization as a part of NEAT genome performance evaluation Razavi et al. (2012a).

With respect to NEAT algorithm implemented in the Encog library (Heaton (2015)) and part of the FeNS system, three code base developments are identified:

A. Encog is currently a multithreaded library efficiently designed and implemented. However, computational scalability on available nodes in a computer cluster needs to be implemented to properly take advantage of CSIP/OMS cloud computing environment potential.

B. Wang et al. (2013) demonstrates that NEAT learning process can be fasten by more accurately numbering ANN nodes. Encog currently implement standard NEAT node numbering by Stanley and Miikkulainen (2002).

C. Feature Deselective NEAT (FD-NEAT) (Tan et al. (2009)) is an alternative feature selection algorithm that has proven to be a promising methodology for classification tasks (Tan et al. (2009)). This methodology might be helpful for developing new gating strategies and should be tested against FS-NEAT. However, this neuroevolutionary algorithm is not part of Encog capabilities yet.

5.2 NET3: CONCLUSION AND FUTURE DEVELOPMENT

Lack of proper software architecture design and application of good programming principles at early stage of model layout obstruct and slow down research model maintenance and development.

Additionally, the need of accounting for more simultaneous physical processes, describing and studying natural phenomena at different scales or introducing innovative engineering design practices drives model development into more capable as well as complex software application. Here, the lack of input/output standard formats, poor documentation and deficient scientific algorithm implementation complicate user approach to a new model, make model learning curve rather steep, and impede user to translate her/his modeling creativity into simulation results eventually.

The introduction of environmental modeling frameworks (EMF)s overall alleviates these issues. However, research community is still reluctant in adopting EMFs as standard model development workflow. Furthermore, actual applications demonstrate that current EMFs modeling capabilities limit modeler creativity anyway.

To overcome these relevant modeling constraints, this dissertation confirms the need of introducing EMFs in standard workflow and support their widespread use by describing the successful case of JGrass-NewAGE/GEOframe system.

Most importantly this dissertation aims to address and alleviate complex research modeling solutions maintenance, development, and application and proposes the extension of EMFs flexibility by integrating a graph modeling structure to elevate the concept of modeling encapsulation and software reusability.

NET3 is graph modeling structure fully integrated in the core of OMS3 and expands OMS3 capabilities by interconnecting different modeling solutions that share common input/output variables. Every modeling solution is encapsulated into a node of the graph modeling structure, and same modeling components can be reused in different nodes avoiding code duplication, error-prone maintenance and development, and thus facilitating composition of complex innovative modeling solution.

NET3 demonstrates an opportunity for research model developers and users to streamline complex modeling solution maintenance and development and to wide open modeler creativity.

NET3 makes use of the river network – graph structure analogy to facilitate composition of network based modeling solution (see Subsection 4.5.1). It allows for setting up different modeling solutions to properly describe physical processes in mountain, hill, and plain subcatchments and connecting them. It allows for finely tuning model input parameters per subcatchment, provides multi-site calibration capabilities.

NET3 enables a further layer of implicit parallelization based on network topology to speed up the overall simulation run-time. Thoroughly setting up modeling solution granularity allows for higher level of computational scalability (see Subsection 4.5.2).

NET3 is a flexible graph modeling structure that can be applied to any network based modeling solution and allows for different iterative loops (temporal or convergence) in each node of the modeling structure. It additionally allows for running a inner graph modeling structure from within each node of a graph modeling structure (see Subsection 4.5.3).

NET3 strictly follows OMS3 design principles (and EMFs more generally) and fully decouples software architectural aspects from scientific concepts. Leveraging NET3 features doesn't require any additional programming proficiency compared to OMS3 requirements.

However this dissertation doesn't investigate every research question proposed and additionally identifies several areas of NET3 improvements.

Since application of graph theory is a very active field of research, the following two questions are soon to be studied:

1. Can NET3 layer of implicit parallelism effectively speed up the computation of both small and large scale modeling solutions?

2. What is the proper trade off between graph topology (NET3) and component connections (OMS3) related parallelizations?

Before investigating these research questions, NET3 has to be redesigned to properly scale on available nodes of a computer cluster network. NET3 implementation is currently based off of multithreading computation. This architectural aspect doesn't allow for deeply leveraging super computing environments.

A *"scalable"* NET3 will allow for properly implementing automated parallel multi-site calibration procedures; finely designing memory management to overcome current limitations when it comes to modeling with Graph of Graphs feature; additionally implementing automated management of enabled/disabled computational branches.

A INCREASING COMPLEXITY

Contents

In Section Issues related to the use of mathematical models on the field, an in-depth analysis of limits and constraints of operational use of mathematical models in consultancy and research environments has been performed. To support the reasons that motivate this research, a further analysis on model complexity with respect to user learning curve is conducted. As expected, mathematical models develops and evolves over time. This might have positively or negatively affected issues related to operational use just described. Surely, software and code base became more and more complex. The evaluation of this increasing complexity along with analysis of positive or negative impact of software usability is an important contribution to this research.

Leveraging available model documentation and open source environmental simulation models, two type of measurements are performed:

1. Number of model input parameters;
2. Software Engineering (SE) metrics.

The number of model input parameters relates to model user's efforts in preparing and preprocessing input model data to properly reflect the study area. It is a means for evaluating model user experience.

SE metrics is a set of measurements and indices to support software engineers evaluation of code base aspects: from software quality, to cost estimate of the entire project, to size/complexity of a software package in order to properly plan future maintenance (Boughton (2011)). As a consequence, software metrics returns objective, reproducible, and quantifiable values of software quality, complexity, and maintainability (Boughton (2011)).

Some of most common software metrics are following listed and briefly described (Wikipedia (2004)):

- **Code coverage:** (or test coverage) estimates the percentage of source code executed during test suite runs. In order to systematically test each portion of the code, input data varies between the range of all possible and significant combinations (Miller and Maloney (1963)). The higher the code coverage the lower the possibility of unexpected software bugs.
- **Cohesion:** is the *"degree to which the elements inside a module belong together"* (Yourdon and Constantine (1979); Stevens et al. (1974)). A class or module highly cohesive (and consequently loosely coupled) has important attributes like robustness, reusability, readability, and reliability.

- **Coupling:** measures how strongly two modules or classes are interconnected. A loosely coupled software (and consequently highly cohesive) usually indicates good software architecture and design.

- **Cyclomatic complexity:** (or McCabe's complexity) evaluates *"the number of linearly independent paths within a section of source code"*. Control flow graph is the ideal representation: *"the nodes of the graph correspond to invisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command"*.

- **Number of Lines of Code (LOC):** counting the number of lines in the source code, it provides a measure of the size of a computer program. Physical LOC accounts for the pure lines of code without comment lines, while logical LOC (or LLOC) tries to estimate the number of executable *statements*.

The software *cloc* Danial (2018) has been used to carry out the following metric analysis. *cloc* is able to estimate blank lines, comment lines, and physical lines of source code. It is also able to evaluate differences between previously listed features when two versions of the same software are provided.

Two environmental models were analyzed with respect to their evolution in source code, comments, and number of model parameters to support the hypothesis of an increasing complexity over time. Those models were selected since they have been developed for more than a decade and have an established user and developer community. Additionally, model source code of early/previous versions was made available for this study by the original model developer or group. Every model is briefly introduced, user and software metrics are obtained and discussed.

A.1 SWAT

Soil and Water Assessment Tool (SWAT) is a semi-distributed, watershed (river basin) scale, physically based model. Its development started in the early 1990s at USDA-ARS in Temple, Texas, led by Dr. J. G. Arnold, and it's an ongoing process still (Devia et al. (2015)). It addresses environmental issues that relate to the evaluation of management practices effects on water resources, sediments transport and nonpoint-source pollution (e.g. agricultural chemical yields) (Arnold et al. (2012)).

SWAT originates from the merging of two models: Simulator of Water Resources in Rural Basin (SWRBB) (Williams et al. (1985); Arnold et al. (1990)) and Routing Outputs to Outlet (ROTO) (Arnold et al. (1995)). SWRBB was a water quality assessment tool. Its applicability constraints were watersheds size (up to few hundred square kilometers) and number (up to 10 subbasins). These constraints became a real modeling issue in the late 1980s, when the Bureau of Indian Affairs required a monitoring tool for evaluating water quality within american native reservations. The targets were Arizona and New Mexico reservation lands which cover a total area of several thousands square kilometers. ROTO was then implemented to handle and connect several outputs from SWRBB runs and properly route them through channels and reservoirs. This temporary solution worked more like a prototype for the proposed problem. SWAT is the final product, a completely redesigned and reimplemented software built from the previous experience of combining SWRBB and ROTO capabilities.

Even without diving into each detail of initial hydrological processes implemented in SWAT, it is recognizable how the merging of two models results into a big core simulation software. The merging process additionally allows for coupling previously independent features (e.g. routing (ROTO) of water quality results estimated in independent watershed (SWRBB)) which originate a more capable as well as complex software code base.

Since the initial merging phase in the early 1990s, significant improvements expanded SWAT capabilities. The most noteworthy functionalities are well described in Williams et al. (2008), and following briefly summarized:

- Addition of management of multiple HRUs;
- Crop managements area adding auto-fertilization and auto-irrigation management options, adding CO_2 evaluation during crop growth for climatic change, improving grazing manure applications;
- Hydrological processes area adding canopy storage, adding Penman-Monteith potential evapotranspiration, improving snow melt routines, adding Green & Ampt infiltration module, adding Muskingum routing method, adding weather forecast scenarios;
- Water quality processes area improving in-stream nutrient water quality equations, in-stream pesticide routing, nutrient cycling routines, adding reservoir/pond/wetland nutrient removal by settling, adding routing of metals, adding bacteria transport algorithms.

To evaluate the potential impact of these new functionalities on modeling code base growth and consequent user experience, source code of four consequential officially released versions of SWAT are analyzed. Downloaded software is available online at `https://swat.tamu.edu/software/swat-executables/` and classified as follow: SWAT2000, SWAT2005, SWAT2009, and SWAT2012 (latest release).

A.1.1 User experience

From a user standpoint, SWAT is a well documented software. Every release is provided with detailed user documentation and some releases (2005 and 2009) are accompanied by two files actually: theory and input/output documentations.

Although available documentation facilitates user approach to the model by providing usage references and requirement descriptions to begin with, learning how to get started with the simulation software might not be a quick and easy task. Considering I/O documentation only, which is common to every software release, the number of pages goes from about 450 in the first release till 650 in the latest.

Furthermore, from a brief analysis of available documentation, user has to deal with an important number of input files: from 33 for v2000 to 37 for v2009 and v2012 (see Figure 95).

The number of input parameters is an important metric that allows for estimating amount of user's data and work required for setting up a modeling simulation. Although model set up (and consequent input parameter requirements) is built upon computational modules enabled for a specific type of modeling simulation, an overall evaluation of number of model input parameters between versions returns an important increment between first two versions (v2000, v2005) (from 540 input parameters to 612) and constant trend over the other version (v2005, v2009, v2012) (see Figure 96).

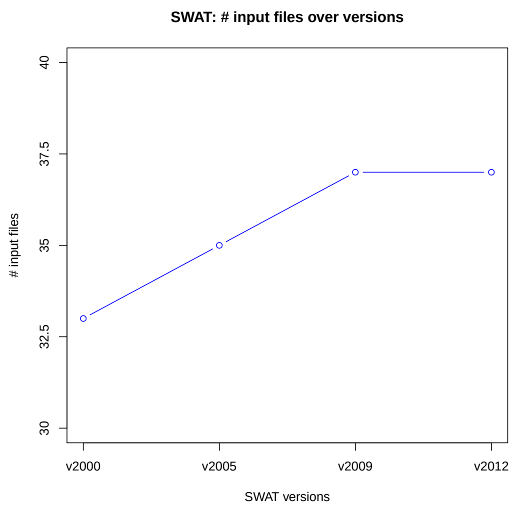


Figure 95: Trend of number of input files required for running a SWAT simulation over official releases.

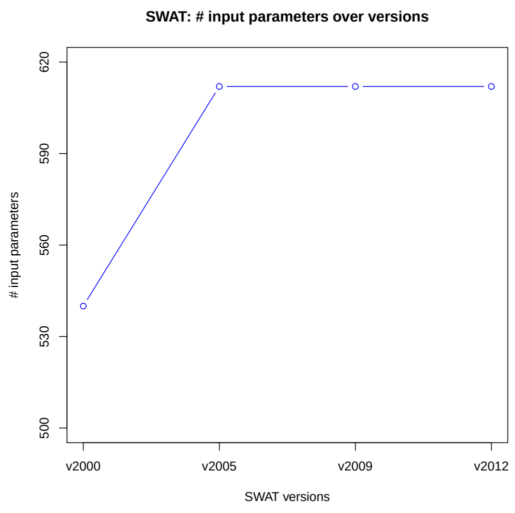


Figure 96: Trend of number of input parameters to setup for running a SWAT simulation over official releases.

Users' Documentation of each model version states that some parameters in several input files are derived using the SWAT GIS tool. Consequently, a model user doesn't really have to provide data or information for each and every input parameter. Additionally, some input files and related parameters are optional and not required for a simple standard SWAT run.

In conclusion, SWAT is a thoroughly documented, heavily tested and validated modeling software. Its development lasts for over two decades and it is an ongoing process still. Several modeling modules and wide range of modeling capabilities are made available and this reflects to important documentation size, input files and input parameters number. Although this last two metrics haven't increased much over software versions, learning how to accurately and correctly make use this sort of model requires study and dedicated knowledge. User approach to SWAT begins with a user manual of about 650 pages and more than 600 available input parameters.

A.1.2 Software metrics

SWAT is Fortran code base. Fortran is a notable general-purpose imperative programming language, which is usually codified into monolithic software applications. This is a drawback when it comes to software maintenance and development. Monoliths are notably close to modifications since their structure is resistant to accommodate new features (Martin (2009); David et al. (2013); Newman (2015); Nadareishvili et al. (2016)). This analysis, however, doesn't investigate complexity with respect to software maintenance and development and narrows its scope to physical lines of source code and comments. *cloc* facilitated the estimate of these two metrics.

Estimated metrics (Figure 97) shows a growing trend for both number of lines of code and comments across available software versions.

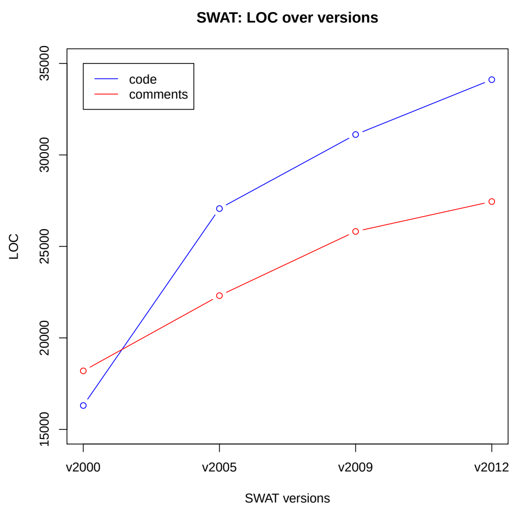


Figure 97: Trend of number of lines of source code (blue line) and comments (red line) in SWAT code base over official releases.

The first SWAT version analyzed (v2000) shows a 1:1 relationship between number of lines of code and comments. This is a positive aspect since means that source code, algorithms, and routines APIs are described.

The positive trend of lines of code and comments over SWAT versions indicates model developer dedications in documenting newly added modules and computing capabilities.

The overall increment of number of lines of code from v2000 to v2012 is more than 100% (Figure 98 and Table 7): the initial computational core was about 16 300 lines and ended up being about 34 100 lines.

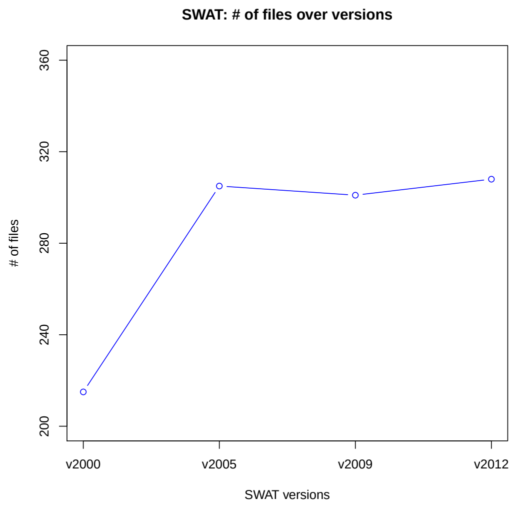


Figure 98: Trend of number of lines of files of the SWAT code base over official releases.

The biggest improvement was between v2000 and v2005. The initial core jumped from 16 300 to 27 000 lines of code with a total increment of around 80% (Figure 97 and Figure 99, Table 8). The number of files increased of 50% (Figure 98) going from 215 to 305. Number of file in the following versions (v2009, v2012) remained almost constant.

Table 7: Number of file, lines of code, and comments of SWAT over official releases.

	SWAT2000	SWAT2005	SWAT2009	SWAT2012
files	215	305	301	308
loc (code)	16 308	27 068	31 114	34 113
loc (comment)	18 199	22 311	25 819	27 452

Around 38% of lines of code (6 174) was added between v2005 and v2009, while 13.4% only (1 173) between v2009 and v2012 (Figure 99 and Table 8). Between these three software versions the amount of modified lines of code is almost constant and around 8% (Figure 99 and Table 8). The negative trend of added lines of code opposes to a positive trend of unaltered lines of code: between v2005 and v2009 68.3% of software code base stays the same while this amount rises up to 88% between v2009 and v2012 (Figure 99 and Table 8).

This analysis demonstrates that number of lines of code added to SWAT code base actually reflects the amount of innovative features listed by Williams et al. (2008). The overall computational core duplicated in size between the first and the last release.

However, a negative trend of added lines of code between versions opposes to a positive trend of unchanged lines of code (Figure 99). This drive to two conclusions:

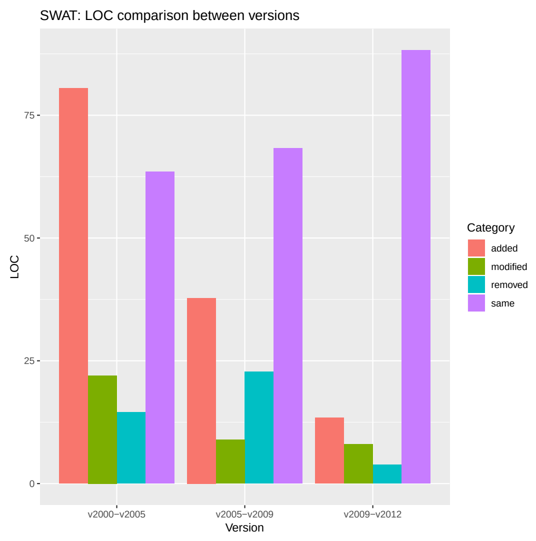


Figure 99: Percentage of number of added, modified, removed, and unchanged lines of code between SWAT consecutive officially released versions. Percentages are computed on the older version.

1. the inner complexity of the simulation model keeps on increasing since new lines of code (and new functionalities consequently) are always added over software releases;
2. SWAT software core base is getting to a point of consolidated robustness and stability since the amount of unmodified lines of code is rising over 88%.

Summarizing, SWAT is a complex software which every release is enriched of new functionalities with. This reflects into an increasing number of lines of code and intrinsic complexity in model maintenance and development consequently. However, the addition rate of new features over releases slows down in favor of robust and consolidated software core.

Table 8: Percentage of identical, modified, removed, and added number of lines of code between SWAT versions (percentage computed on the older version).

	v2000–v2005		v2005–v2009		v2009–v2012	
same	10 355	63.5%	18 493	68.3%	27 436	88.2%
modified	3 588	22%	2 401	8.9%	2 505	8%
removed	236	14.5%	10 220	22.8%	4 172	3.8%
removed	13 125	80.5%	6 174	37.8%	1 173	13.4%

A.2 SWMM

Storm Water Management Model (SWMM) is a computational model for estimating quantity and quality of urban runoff. Its application ranges from modeling of single event to long-term simulations. It is broadly used especially for stormwater and sanitary sewer design, analysis of pollutant transport, treatment strategies of point and nonpoint sources, and urban planning (Gironás et al. (2010)).

Two main modules constitute the whole software: the runoff component and the routing component. The first one operates on a subcatchment scale: precipitation falling on each subcatchment generates runoff and pollutant load. The second component routes the generated runoff throughout a network of pipes, channels and several type of facilities and instruments like storage/treatment devices, pumps, and regulators.

The development history of this notable software started in 1971 (Rossman (2010)). SWMM I is the first software release and was developed by Metcalf & Eddy, Inc., Water Resources Engineers, and University of Florida. Since then, four more official major version have been released: SWMM II (1975, contributor University of Florida) which is the actual first widely distributed release; SWMM 3 (1981, contributors University of Florida, and Camp Dresser & McKee) where full dynamic wave flow algorithm, Green-Ampt infiltration, snow melt and continuous simulation were added; SWMM 3.3 (1983, contributor US EPA) which is the first PC version ever released (previous versions are paper only); and SWMM 4 (1988, contributors Oregon State University and Camp Dresser & McKee) where groundwater, RDII, and irregular channel cross-sections were introduced.

SWMM 5 is the latest stable version. Initially released in 2005, it has been actively developed, supported and released on a regular bases in the last 14 years. US EPA and CDM-Smith are the main contributors. Several new features were added throughout its development. However, the most important advancements relate to complete translation and refactoring of the core engine from Fortran language to C language and the addition of a graphical user interface.

The analysis of SWMM development history includes SWMM5 only. The source code of previous versions is not available online. However, SWMM5 accounts for 24 official releases over 15 years. The analyzed source code was downloaded from openswmm website and USEPA website.

A.2.1 User experience

User documentation contains detailed and exhaustive explanations for preparing a SWMM modeling solution.

A comprehensive and well developed GUI smooths user's very first approach to SWMM. It also facilitates input data and parameters management and set up.

However, user has to recursively edit the project file to design the sewer system. This is an acceptable workflow when limited number of sewer pipes, facilities, and instruments are involved. The process becomes complicated and overwhelming in case of larger project files stored in CAD or GIS formats. Here, user might have to manually set up input parameters for each modeling object such as network nodes or links.

Software modeling input objects and parameters are following briefly summarized.

SWMM allows for modeling processes of four water management related categories: hydrology, hydraulics, water quality, and treatment. The overall number of modeling objects is 20. Every modeling object can be finely tuned from within the project file, which is split into 44 different sections.

111 generic modeling simulation parameters are provided. The other parameters are dedicated to describe the modeling behaviour of each element of the sewer network: 8 parameters describe rain gauge behaviour; 67 characterize sub catchment while 8 represent subareas responses; 19 parameters are required for LID design; 14 characterize an aquifer; 6 a junction, additional 6 an outfall; 11 parameters

represent the divider behaviour; 13 for a storage; 9 for a conduit; 7 for a pump; 8 for an orifice; 12 for a weir; 8 for an outlet; 19 for a cross-section; 6 head-loss; 31 pollutant; 27 per node.

The overall number of input parameters is 382 accounting for a network with one element per category (Rossman (2010)).

These numbers surely express high level of software flexibility and adaptability to modeler requirements. They intrinsically indicate important amount of study and work to (1) understand how model operates, and (2) actually provide and set up input parameters for a modeling simulation run.

Additional analysis and comparison of modeling objects and input parameters between software version are not since only one user manual is provided and generically refers to SWMM5.

In conclusion, SWMM is a flexible and highly tunable modeling software that makes available a large number of modeling objects and input parameters. This indicates a steeper learning curve to finely set up and exercise a modeling simulation. Furthermore, the addition of innovative engineering solutions to water management problems will intrinsically reflect into an increasing number of modeling object and input parameters consequently.

A.2.2 Software metrics

SWMM5, previously coded in Fortran, was completely redesigned for the 2005 release and is currently C code base. C is a notable general-purpose imperative programming language, which is usually codified into monolithic software applications.

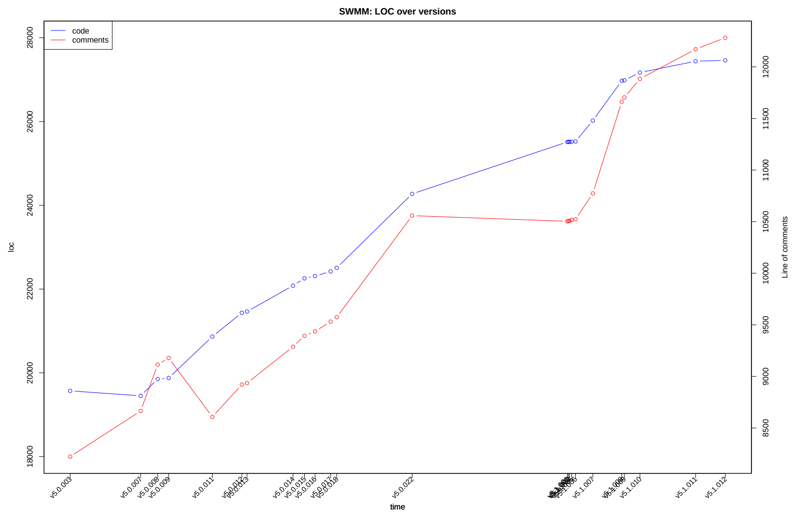


Figure 100: Trend of number of lines of source code (blue line) and comments (red line) in SWMM code base over officially released versions. Left Y axis shows the magnitude of the number of lines of code, right Y axis shows the magnitude of the number of lines of comments.

Analysis over time of number of lines of code and comments in the SWMM5 official releases illustrates positive and similar trends (Figure 100). This demonstrates developers dedication in regularly commenting newly added modeling capabilities and algorithms.

The total number of lines of code goes from 19 570 for version v5.0.003 to 27 461 for version v5.1.012 (Figure 100 and Table 9). This is a reasonable development over 14 years (+40%). It indicates software maturity and stability as well as consistent implementation of new modeling features to accommodate innovative water management related engineering methodologies. As a matter of fact, Figure 101 shows only percentages of added, modified and removed number of lines of code since 90% of software core doesn't change in between released versions.

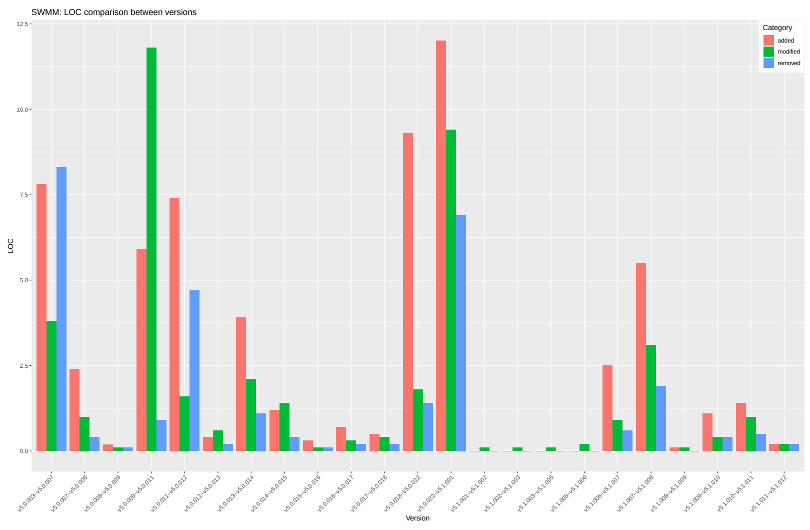


Figure 101: Percentage of number of added, modified, and removed lines of code between SWMM consecutive officially released versions.

Trend in Figure 101 mostly shows little adjustments and improvements of software modeling capabilities and performance in between software releases such as (1) management of lateral groundwater flow to groundwater module; (2) encapsulation of algorithms for solving the momentum equation of dynamic wave flow routing in sewer pipes to improve computational parallelization; (3) addition of routines to evaluate conduit water evaporation and infiltration rate; (4) addition of routines for estimating LID hydrologic performance; (5) code refactoring and application of software design principles.

In conclusion, SWMM is a well developed and maintained modeling software. The overall source code grew up of 40% in about 14 years. This is a plausible development rate, which demonstrates a regular implementation of new engineering modeling practices and software release consequently. The increasing number of lines of source code (and modeling features consequently) intrinsically reflects into a growing complexity in model maintenance, development, and usage.

Table 9: Number of file and lines of code of SWMM code base over official releases.

SWMM version	Loc (code)	Loc (comment)
v5.0.003	19 570	8 223
v5.0.007	19 451	8 665
v5.0.008	19 852	9 115
v5.0.009	19 878	9 179
v5.0.011	20 866	8 608
v5.0.012	21 434	8 921
v5.0.013	21 464	8 935
v5.0.014	22 081	9 288
v5.0.015	22 257	9 394

(...continue to next page)

SWMM version	Loc (code)	Loc (comment)
v5.0.016	22 310	9 436
v5.0.017	22 421	9 531
v5.0.018	22 508	9 574
v5.0.022	24 273	10 558
v5.1.001	25 513	10 503
v5.1.002	25 515	10 506
v5.1.003	25 515	10 508
v5.1.005	25 518	10 519
v5.1.006	25 524	10 523
v5.1.007	26 023	10 774
v5.1.008	26 973	11 662
v5.1.009	26 985	11 704
v5.1.010	27 171	11 884
v5.1.011	27 440	12 171
v5.1.012	27 461	12 282

Table 10: Percentage of modified, removed, and added number of lines of code between SWMM versions (percentage computed on the older version).

SWMM versions	modified		added		removed	
v5.0.003-v5.0.007	734	3.8%	1 518	7.8%	1 637	8.3%
v5.0.007-v5.0.008	190	1%	478	2.4%	77	0.4%
v5.0.008-v5.0.009	13	0.1%	36	0.18%	10	0.1%
v5.0.009-v5.0.011	2 341	11.8%	1 168	5.9%	180	0.9%
v5.0.011-v5.0.012	326	1.6%	1 545	7.4%	977	4.7%
v5.0.012-v5.0.013	130	0.6%	79	0.4%	49	0.2%
v5.0.013-v5.0.014	443	2.1%	847	3.9%	230	1.1%
v5.0.014-v5.0.015	311	1.4%	260	1.2%	84	0.4%
v5.0.015-v5.0.016	21	0.1%	72	0.3%	19	0.1%
v5.0.016-v5.0.017	69	0.3%	153	0.7%	42	0.2%
v5.0.017-v5.0.018	83	0.4%	123	0.5%	36	0.2%
v5.0.018-v5.0.022	39	1.8%	2 087	9.3%	322	1.4%
v5.0.022-v5.1.001	2 272	9.4%	2 911	12%	1 671	6.9%
v5.1.001-v5.1.002	6	0.1%	3	0%	1	0%
v5.1.002-v5.1.003	12	0.1%	0	0%	0	0%
v5.1.003-v5.1.005	13	0.1%	11	0%	8	0%
v5.1.005-v5.1.006	59	0.2%	8	0%	2	0%
v5.1.006-v5.1.007	240	0.9%	645	2.5%	146	0.6%
v5.1.007-v5.1.008	815	3.1%	1 422	5.5%	472	1.9%
v5.1.008-v5.1.009	12	0.1%	13	0.1%	1	0%
v5.1.009-v5.1.010	117	0.4%	294	1.1%	108	0.4%
v5.1.010-v5.1.011	276	1%	393	1.4%	124	0.5%
v5.1.011-v5.1.012	61	0.2%	62	0.2%	41	0.2%

B R AND PYTHON ANNOTATION BINDINGS FOR OMS

Contents

OMS3 is an environmental modeling framework designed to support and simplify the development of scientific environmental models. It is implemented in Java, a programming language that allows the framework to be flexible and non-invasive. Consequently, Java is the native language for developing OMS-compliant components. However, OMS3 aims to ensure the longevity of old model implementations by providing C/C++ and Fortran bindings that allow for connecting slightly modified legacy environmental software to newly developed Java components. In the recent years, three scientific programming languages drew the modeling community's attention: R, Python, and NetLogo. They have a flat learning curve, numerous scientific libraries, and duck typing makes them an attractive solution for fast scripting. Furthermore, they have an active developer community that keep releasing and improving open source scientific packages. This is a relevant aspect when it comes to facilitating and speeding up the implementation of scientific algorithms. Therefore, OMS3 integration capabilities have recently been enhanced to provide R, Python, and NetLogo bindings. As a result, multi-language modeling solutions can be tailored to meet the scientific community's needs. Thanks to the framework's non-invasiveness, R, Python and NetLogo scripts must only be slightly modified with source code annotations to become OMS-compliant components. The resulting components are nevertheless still executable from within the original environments. This contribution shows two actual applications of the implemented R and Python bindings, the NetLogo implementation is not addressed in this paper. The Regional Urban Growth (RUG) is implemented in R and the TRansportation ANalysis SIMulation System (TRANSIMS) models require the Python Run Time Environment (RTE) module to run. The RUG model is a landscape model capable of evaluating impacts of new regional urban development on surrounding environment and projecting long-term growth-management plans. TRANSIMS is a software suite based on a cellular automata microsimulator which performs regional transportation system analyses. Both model suites are among OMS enabled models for the FICUS project, the *"Framework for Integrating the Complexity of Uncertain Systems"*. Furthermore, the model application flexibility was enhanced by introducing Docker containers in the workflow to alleviate the burden of complex software management and setup.

B.1 INTRODUCTION

OMS3 is a flexible and non-invasive environmental modeling framework (David et al. (2013); Lloyd et al. (2011)). Its main objective is to simplify environmental model development by streamlining the translation of physical processes into programming algorithms. It allows for encapsulating each algorithm into a standalone component ensuring the *single responsibility* principle. It lowers the development effort related to data reading and writing, data analysis and visualization, component interaction, temporal-spatial stepping and multi-threading/multi-processor computations. As a result scientists can focus on scientific understanding of environmental phenomena rather than software development.

OMS3 is Java-based, and therefore Java components are natively supported. In order to maintain compatibility with legacy Fortran and C/C++ software, OMS3 makes use of native shared libraries and provides Fortran and C/C++ bindings. However, the modeling community's use of scripting/programming languages like R, Python and NetLogo is rapidly taking off. These languages are easy to learn and use because of their friendly syntax and semantics. They rely on user and developer communities, which share on-line implementation and problems solutions, generic information and most importantly well designed scientific packages. Some notable Python examples are NumPy (Oliphant (2006)) and SciPy (Jones et al. (2014)). Some notable R examples are gstat (Pebesma and Wesseling (1998)), raster (Hijmans et al. (2015)) and randomForest (Liaw et al. (2002)).

Scientists and engineers solely want to focus on solving their research questions and problems. These scripting languages are consequently very attractive and proper OMS3 bindings have become necessary. The main concern while developing OMS bindings was to keep the user experience in setting up Python and R OMS-compliant components as close as possible to OMS Java component development. Section B.2 is focused on describing the user approach in modifying Python and R scripts into OMS-compliant components. Section B.3 describes actual framework side implementation of both bindings while section B.4 introduces the process of bundling OMS3 into a Docker image. Section B.5 shows two actual applications: the R-based Regional Urban Growth (RUG) model (Westervelt et al. (2011)) and the Python wrapped TRansportation ANalysis SIMulation System (TRANSIMS) model (Smith et al. (1995)). Section B.6 provides concluding remarks and identifies current constraints and needs for future development.

Moreover, OMS3 was recently bundled into a Docker (Merkel (2014)) image to further simplify user experience: once Docker is installed on the machine, no further software installation and library linking are required to run OMS3. A user needs to provide only a properly set up OMS3 project. The Docker container then takes care of building the project and running the modeling solution, automatically connecting every type of component.

B.2 USER EXPERIENCE

An OMS component is basically a plain Java class with framework metadata annotations. Input/output variables are listed as fields and annotated with `@In` and `@Out` OMS annotations. The one mandatory method with an `@Execute` annotation encapsulates the main algorithm and calls related methods or objects. Two more methods can be annotated with `@Initialize` and `@Finalize` and are respectively executed before and after the entire simulation. They are optional methods, though. A user may also add further optional annotations to capture comments and component design ideas into metadata for generating documentation later or perform tests.

These basic concepts were used in the design of both Python and R bindings. Accordingly, two main development steps were identified to seamlessly adapt Python or R scripts into OMS-compliant components:

1. Determine the function encapsulating the main algorithm if the script is already split into functions, otherwise wrap the entire script into one main function;

2. Identify input and output variables and list them at the very beginning of the script.

Then, suitable annotations have to be accomodated. Listing B.1 and Listing B.2 ease the understanding of this simple but crucial step: Listing B.1 shows the annotated code snippet of the R component `AttractorAnalysis.R`, which is part of the RUG model; Listing B.2 illustrates the annotated code snippet of the Python component `TransimsObj.py`, which is the Python wrapper for executing and connecting TRANSIMS executables.

A couple of similarities can be underlined in Listing B.1 and Listing B.2: annotations are hidden in comments; Java data types are explicitly specified right after `@In` and `@Out` annotations.

The first aspect allows for maintaining compatibility of scripts with their original interpreters. To execute the OMS-compliant scripts from within their original environments, user is asked to: (1) assign input values to each input variable and null values to each output variable (or just comment them to avoid parsing errors); (2) call the main function to execute the script.

The second aspect takes into account the absence of declared data types in both Python and R. Thus, Java equivalent types must be defined between parentheses right after the annotation to allow for proper conversions when R or Python components are connected to Java or Fortran or C/C++ components.

Listing B.1 shows how a stack of raster maps (`masterRaster`, line 5), a list of raster maps (`interconnectMaps`, line 8) and a list of strings (`instructions`, line 11) are fed to the `AttractorAnalysis.R` component. After the proper computation, the raster map describing the attractiveness of strategic locations in the study area (`attractorMap`, line 14) is returned. Listing B.2 shows how the path to the directory gathering TRANSIMS modules (`BINDIR`, line 6), the path to the working directory (`PROJECT`, line 8), the name of the TRANSIMS module (`executable`, line 10) and name of the related file of input data and parameters (`controlFile`, line 12) are inputs to the `TransimsObj.py` component. When the run is over, the component returns the proper message (`simDone`, line 14).

Listing B.1: R OMS-compliant version of the `AttractorAnalysis.R`

```
1  library(raster)
2  library(doParallel)
3
4  # @In("CoverageStack")
5  masterRaster
6
7  # @In("List<GridCoverage2D>")
8  interconnectMaps
```

```
 9
10  # @In("List<String>")
11  instructions
12
13  # @Out("GridCoverage2D")
14  attractorMap
15
16  # @Execute
17  main <- function() {
18    # RUG attractor analysis
19    # ...
20    attractorMap <<- calcAttractorMap()
21  }
```

Listing B.2: Python OMS-compliant version of the `TransimsObj.py`

```
 1  import os
 2  import sys
 3  from TransimsRTE import *
 4
 5  # @In("String")
 6  BINDIR
 7  # @In("String")
 8  PROJECT
 9  # @In("String")
10  executable
11  # @In("String")
12  controlFile
13  # @Out("String")
14  simDone
15
16  # @Execute
17  def execute():
18    # Transims OMS object
19    # ...
20    global simDone
21    simDone = "Transims obj processed"
```

Currently, only standard data type matching is available[1]. The R binding temporarily provides an inner matching of `Raster`, `List of Raster` and `CoverageStack` between the raster R package and the Geotools Java library. However, a plug-in system of data type conversions is under development. The purpose is to allow each user to implement the proper conversion between data types, and sharing it with the entire community. Nevertheless, two connected R or Python components can share generic Object data type which does not require any matching (see Table 11 for available data types conversions).

One design aspect relates to both bindings: in order to actually fill output variables and avoid declaring local function variables, a user must make use of specific operators. In R scripts, output variables must be assigned using the double arrow assignment operator <<− which allows for modifying variables in a parent level (e.g. `attractorMap` in line 20, Listing B.1). In Python scripts, output variables must be declared `global` at the very beginning of the main function to allow for modifying variables at parent level (e.g. `simDone` at line 20, Listing B.2). Output variables have to be declared outside the main function as well (e.g. line 14, Listing B.1 and Listing B.2). In this way, OMS3 can access their content and perform proper connections with other components.

With respect to framework invasiveness, no specific OMS3 or other APIs have to be imported or extended.

1 Python binding makes use of jarray instead of Numpy data structures because jarray makes data transfer faster and more efficient for the back-end Jep.

Table 11: R and Python available data types.

Java data type	R data type	Python data type
int	int	int
double	double	double
String	String	String
int[ ]	vector of int	jarray(...,JINT_ID,...), from jep import jarray, JINT_ID
double[ ]	vector of double	jarray(...,JDOUBLE_ID,...), from jep import jarray, JDOUBLE_ID
String[ ]	vector of String	jarray(...,JSTRING_ID,...), from jep import jarray, JSTRING_ID
GridCoverage2D	raster	
CoverageStack	RasterStack	
List<GridCoverage2D>	list( ) of Raster	
Object	Object	Object
List<Integer>		[ ]
List<Double>		[ ]
List<String>		[ ]
List<Object>		[ ]
Map<Object, Object>		dictionary

B.3 TECHNICAL APPROACH AND IMPLEMENTATION

To provide for a smooth user experience the actual implementation burden is moved into the framework. Python and R are both cross-platform, interpreted, high-level scripting and programming languages. Thus, they both require interpreters to parse and execute a script. Simple access through shared libraries like Fortran or C/C++ through JNI (Gordon (1998)) does not work. Consequently, OMS3 needs to directly intercommunicate to R and Python interpreters.

The common approach implies the generation of a Java OMS component aiming to wrap a single R or Python script while building the OMS3 project. Eventually OMS3 calls only Java classes. When it is time to run the Java wrapper, this starts a connection to R or Python environment, sends the script to get parsed by the proper interpreter, sends input data and retrieves output information. It provides also for properly converting input/output standard data types or data structures between languages. Obviously, R and Python environments have to be already installed on the machine and correctly linked to OMS3.

B.3.1 R back-end: Rserve

Rserve is a TCP/IP server developed by Urbanek (2003) to take advantage of benefits of R functionalities from within different programming languages. It was developed following three important design principles: separation of the R system from the application, flexibility for leveraging most R facilities and speed to have a performant client-server communication. However, the most interesting feature is the management of multiple clients simultaneously. Rserve creates a different data space and working directory for each new connection. Because each R OMS-compliant component opens a new independent connection to the R environment, multiple R OMS-compliant components can be executed in parallel without interference. This allows for leveraging OMS3 implicit multithreading computation.

Rserve requires installation of an R interpreter and the Rserve package on a local computer to properly work with OMS3.

B.3.2 Python back-end: Jep

Jep is an open source Python package (`https://github.com/ninia/jep`) that utilizes both JNI and CPython API to run a Python interpreter from within the JVM. Its main feature is that of creating a different sandboxed sub-interpreter for each new Jep instance. In this way concurrent sub-interpreters don't share imported modules or global variables, thus avoiding conflicts.

To properly exercise Jep from within OMS3, the Jep package has to be installed in addition to the proper Python interpreter. This is not trivial on Windows OS which requires an additional installation of a dedicated build tool. Furthermore, Jep shared libraries have to be accurately linked to the correct environment variable (e.g. `LD_LIBRARY_PATH`) to be accessible by the Java process. Switching between Python2 and Python3 might be confusing and error prone as well.

B.4 DOCKER IMAGE BUNDLE

As explained in subsections R back-end: Rserve and Python back-end: Jep, Rserve and Jep require installation of a proper R or Python environment and accurate linking of involved libraries, Jep especially. But this means that a user is expected to take care of software installation and required libraries, which are both OS specific and require some OS proficiencies. This is diametral to the OMS3 principle of simplifying user experience by separating responsibilities between users and software developers. To overcome this constraint a recently released technology has been leveraged and OMS3 has been bundled into a Docker image.

Docker is a software system that packages a software application and its dependencies into an image. It then runs that image as a virtual container on top of a host OS. It is similar to a virtual machine (VM) since it isolates the running process of bundled applications from interfering with running processes of the host OS. However, container virtualization is more lightweight than a Hypervisor based VM. It virtualizes at operating-system-level without the needs of a hypervisor, which is an additional software on top of the host OS to create, run and manage virtual machines(Merkel (2014)). Docker images are platform independent. Consequently, the same Docker containers run on every OS once Docker is properly installed.

A Docker image results from a build process that starts off from a Dockerfile. The latter contains instructions required to install and setup applications along with dependencies. It also contains instructions for proper library linking and environment variables set up. The latter don't interfere with environment variable of the host operating system because Docker isolates the bundled application from the hosting OS. To correctly exercise an OMS modeling solution the OMS project is mounted into the running Docker container. The OMS3 image is made available at `https://hub.docker.com/r/omslab/oms/` and Dockerfiles are made available at `https://github.com/sidereus3/oms-docker`.

Both, R and Python rely on hundreds of packages which cannot be included into a Docker image for the sake of size limits and the impossibility of continuous updates when new packages are released. To overcome this constraint two slightly different approaches have been implemented.

B.4.1 OMS R packages management

User scripts normally import standard and locally installed R packages through the `library()` command. The Docker image manages linking of bundled R environment to the additional `Rlibs/build/` folder.

The OMS Docker image provides a feature that allows for automatically downloading and building R packages required by R scripts in the OMS project. This is a one-time process which is enabled during OMS project build. When specific R packages are required, the user is asked to create a Rlibs folder inside the main OMS project. The user has to provide a file named package.txt with a list of names of required packages, located in `Rlibs`. During the building step, the Docker container looks for `Rlibs` folder. If it exists and contains the file **`package`**`.txt`, the container reads all the listed packages and builds the dependency tree. Then it starts downloading source code of each package into `Rlibs/source/` creating a local R package repository. As a final step, the Docker image goes through the repository, and builds and installs each package into `Rlibs/build/`.

Because Docker is platform independent the OMS project can be zipped and moved to a different machine. If the version of the Docker image does not change, the transferred OMS projects can be directly executed.

B.4.2 OMS Python packages management

The OMS Docker image does not currently provide any tool for automatically downloading required python packages and related dependencies. However, if the user provides Python packages within the folder `Pylibs/` in the main project directory, the OMS Docker image automatically makes new modules and packages available for standard import.

B.5 APPLICATIONS

The development of both R and Python binding has been continuously tested with two actual models in order to gain experience and drive the development direction from the very beginning. The R binding was tested using the RUG model, while the Python binding was tested with the TRANSIMS model.

B.5.1 RUG model

The Regional Urban Growth (RUG) model evaluates the attractiveness of a specific location with respect to urban growth. It is a raster based model: input data is a landscape raster map which allows for estimating development attraction on each location depending on proximity to development attractors (roads, highways, etc.) (Westervelt et al. (2011)). The RUG model was a stand-alone, well implemented R software that takes advantage of availability of R packages like raster (Hijmans et al. (2015)), doParallel (Calaway et al. (2015)), randomForest (Liaw et al. (2002)) and gdistance (Van Etten (2012)). To make this model OMS-compliant, it was split into three different components: Travel Time Analysis, Development Analysis and Attractor Analysis. This partitioning made possible to identify functions containing the main algorithms and input/output data.

The RUG model performs a raster based analysis, and thus two Java components for raster reading and writing were implemented leveraging Geotools APIs. Proper mappings for `Raster`, `List of Rasters` and `CoverageStack` data structures between Java and R (and vice versa) were included in the R binding. The final modeling solution is illustrated in Figure 102.

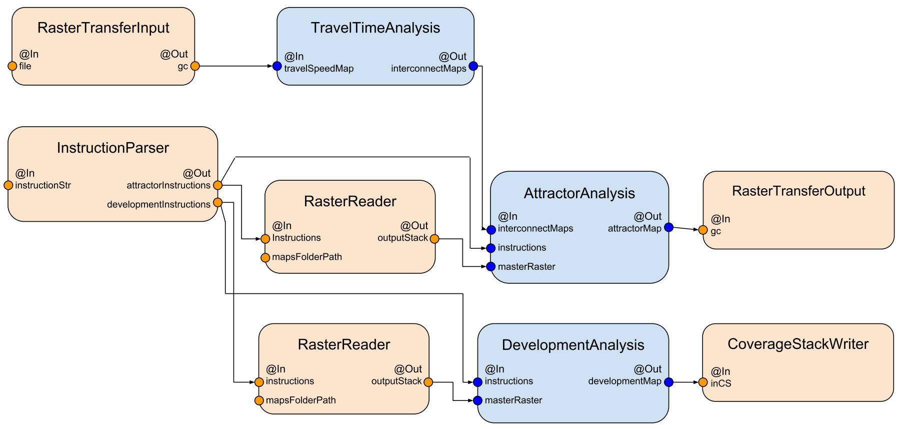


Figure 102: RUG modeling solution: Java components in light orange, R OMS-compliant components in light blue.

B.5.2 TRANSIMS model

The TRansportation ANalysis and SIMulation System (TRANSIMS) model evaluates integrated regional transportation systems. Regional population of individual travelers and freight loads with travel activities and travel plans are core of modeling computation (Smith et al. (1995)). TRANSIMS is more a set of tools than a homogeneous model. Each module is a stand-alone C++ program, which builds into a separate, statically linked executable.

A Python Module for encapsulating TRANSIMS executables has been recently released. TRANSIMS RTE improves scripting flexibility providing for easy modeling solution design. It allows for setting up TRANSIMS keywords, e.g. `@NEW` and `@OLD`, and running a proper executable and related control file from within a Python script. Because a TRANSIMS modeling solution is a sequence of calls to different modules, a generic TRANSIMS-OMS component has been abstracted from a Python script. A simple Java class reads a csv file with a list of executable names and related control files and the feeds the TRANSIMS-OMS component while the list is empty. A sample modeling solution is shown in Figure 103.

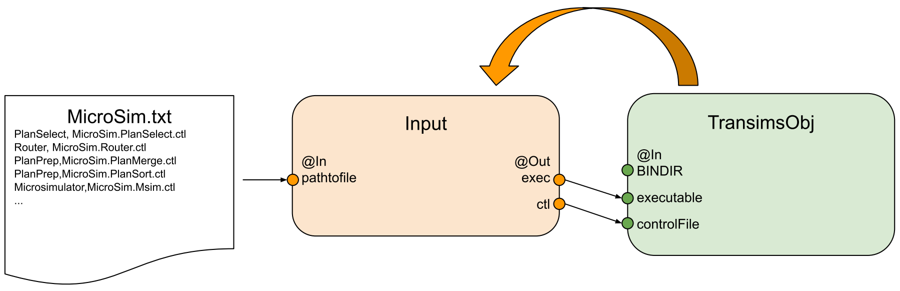

MicroSim.txt
PlanSelect, MicroSim.PlanSelect.ctl
Router, MicroSim.Router.ctl
PlanPrep,MicroSim.PlanMerge.ctl
PlanPrep,MicroSim.PlanSort.ctl
Microsimulator,MicroSim.Msim.ctl
...
Input
@In
pathtofile
@Out
exec
ctl
TransimsObj
@In
BINDIR
executable
controlFile

Figure 103: TRANSIMS sample modeling solution: Java component in light orange, Python OMS-compliant component in light green.

B.6 CONCLUSIONS

This paper shows how two of the most notable and widely used programming languages in the scientific community have been integrated into OMS3. It can be concluded that the process of Python/R scripts adaptation into OMS-compliant components is straightforward and doesn't require user specific proficiency in understanding mixed language programming. This opens a future perspective for easily creating multi-language modeling solutions, that implement againts already available scientific packages and avoid code duplication.

Thank to the innovative technology of Docker containers, a user does not experience the burden of connecting OMS3 with Python and R interpreters. An automated process for R package retrieval and building is provided in the Docker image. The two presented applications demonstrate the applicability and relevance. The implementation aims for design consistency with existing annotation based representation of components.

However, some limitations still exist and will be addressed in future developments: a fully flexible mapping of R/Python into Java data structures is not yet available; automated process for Python packages retrieval is not provided; and only the latest version of a deployed R package is retrieved, user cannot automatically download a specific package version.

BIBLIOGRAPHY

Abera, W., Antonello, A., Franceschi, S., Formetta, G., and Rigon, R. (2014). The udig spatial toolbox for hydro-geomorphic analysis. *Geomorphological Techniques*, 2(4.1):1–19.

Abera, W., Formetta, G., Borga, M., and Rigon, R. (2017a). Estimating the water budget components and their variability in a pre-alpine basin with jgrass-newage. *Advances in water resources*, 104:37–54.

Abera, W., Formetta, G., Brocca, L., and Rigon, R. (2017b). Modeling the water budget of the upper blue nile basin using the jgrass-newage model system and satellite data. *Hydrology and Earth System Sciences*, 21(6):3145–3165.

Akinmolayan, A., Adepoju, K., Adelabu, S., and Osunmadewa, A. (2018). Estimating potential annual soil loss of watershed in nigeria using rulse in a gis and remote sensing environment. In *IGARSS 2018-2018 IEEE International Geoscience and Remote Sensing Symposium*, pages 7504–7507. IEEE.

Allamaraju, S. (2010). *Restful web services cookbook: solutions for improving scalability and simplicity.* " O'Reilly Media, Inc.".

Anees, M., Abdullah, K., Nawawi, M., Norulaini, N., Syakir, M., and Omar, A. (2018). Soil erosion analysis by rusle and sediment yield models using remote sensing and gis in kelantan state, peninsular malaysia. *Soil Research*, 56(4):356–372.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks*, 5(1):54–65.

Apostolopoulos, T. K. and Georgakakos, K. P. (1997). Parallel computation for streamflow prediction with distributed hydrologic models. *Journal of Hydrology*, 197(1-4):1–24.

Argent, R. M. (2004). An overview of model integration for environmental applications—components, frameworks and semantics. *Environmental Modelling & Software*, 19(3):219–234.

Arnold, J. G. and Allen, P. M. (1999). Automated methods for estimating baseflow and ground water recharge from streamflow records 1. *JAWRA Journal of the American Water Resources Association*, 35(2):411–424.

Arnold, J. G., Moriasi, D. N., Gassman, P. W., Abbaspour, K. C., White, M. J., Srinivasan, R., Santhi, C., Harmel, R., Van Griensven, A., Van Liew, M. W., et al. (2012). Swat: Model use, calibration, and validation. *Transactions of the ASABE*, 55(4):1491–1508.

Arnold, J. G., Williams, J., Nicks, A., Sammons, N., et al. (1990). Swrrb; a basin scale simulation model for soil and water resources management. *SWRRB; a basin scale simulation model for soil and water resources management.*

Arnold, J. G., Williams, J. R., and Maidment, D. R. (1995). Continuous-time water and sediment-routing model for large basins. *Journal of Hydraulic engineering*, 121(2):171–183.

Ascough, J., Green, T., David, O., Kipka, H., McMaster, G., Fink, M., Krause, P., and Kralisch, S. (2014). Advances in distributed watershed modeling: A review and application of the agroecosystem-watershed (ages-w) model. In *7th Intl. Congress on Env. Modelling and Software. International Environmental Modelling and Software Society (iEMSs): San Diego, CA, USA, this issue*.

Ascough II, J., David, O., Krause, P., Heathman, G., Kralisch, S., Larose, M., Ahuja, L., and Kipka, H. (2012). Development and application of a modular watershed-scale hydrologic model using the object modeling system: Runoff response evaluation. *Transactions of the ASABE*, 55(1):117–135.

Ascough II, J. C., Green, T. R., David, O., Kipka, H., and MACMASTER, G. (2015a). The spatially-distributed agroecosystem-watershed (ages-w) hydrologic/water quality (h/wq) model for assessment of conservation effects. In *Annual Hydrology Days Conference Proceedings*, pages 23–25.

Ascough II, J. C., Green, T. R., David, O., Kipka, H., McMaster, G. S., and Lighthart, N. P. (2015b). The agroecosystem-watershed (ages-w) model: Overview and application to experimental watersheds. In *2015 ASABE Annual International Meeting*, page 1. American Society of Agricultural and Biological Engineers.

Asher, M. J., Croke, B. F., Jakeman, A. J., and Peeters, L. J. (2015). A review of surrogate models and their application to groundwater modeling. *Water Resources Research*, 51(8):5957–5973.

Bancheri, M. (2017). *A flexible approach to the estimation of water budgets and its connection to the travel time theory*. PhD thesis, University of Trento.

Bancheri, M., Abera, W., Rigon, R., Formetta, G., David, O., and Serafin, F. (2015). Implementing a travel time model for the entire river adige: the case on jgrass-newage. In *AGU Fall Meeting Abstracts*.

Bancheri, M., Formetta, G., Serafin, F., Rigon, R., Green, T. R., and David, O. (2016). Replicability of a modelling solution using newage-jgrass. In *International Environmental Modelling and Software Society (iEMSs), Tolosa, France*.

Bancheri, M., Mita, L., Viaggiano, D., and Manfreda, S. (2018a). Geoframe-newage: a web-based early warning decision support system. In *EGU General Assembly Conference Abstracts*, volume 20, page 16526.

Bancheri, M., Rigon, R., Serafin, F., Abera, W., and Bottazzi, M. (2017a). Strategies for estimating the water budget at different scales using the jgrass-newage system. In *AGU Fall Meeting Abstracts*.

Bancheri, M., Serafin, F., Bottazzi, M., Abera, W., Formetta, G., and Rigon, R. (2018b). The design, deployment, and testing of kriging models in geoframe with sik-0.9. 8. *Geoscientific Model Development*, 11(6):2189–2207.

Bancheri, M., Serafin, F., Formetta, G., Rigon, R., and David, O. (2017b). Jgrass-newage hydrological system: an open-source platform for the replicability of science. In *EGU General Assembly Conference Abstracts*, volume 19, page 17109.

Bancheri, M., Serafin, F., and Rigon, R. (2018c). Travel-time (tt) based modelling of transport in hydrological systems. In *Integrated Hydrosystem Modelling*.

Bancheri, M., Serafin, F., and Rigon, R. (2019). The representation of hydrological dynamical systems using the extended petri nets (epn). *Water Resources Research*.

Band, L. E. (1986). Topographic partition of watersheds with digital elevation models. *Water resources research*, 22(1):15–24.

Bayramov, E., Schlager, P., Kada, M., Buchroithner, M., and Bayramov, R. (2019). Quantitative assessment of climate change impacts onto predicted erosion risks and their spatial distribution within the landcover classes of the southern caucasus using gis and remote sensing. *Modeling Earth Systems and Environment*, pages 1–9.

Beasley, D., Huggins, L., and Monke, a. (1980). Answers: A model for watershed planning. *Transactions of the ASAE*, 23(4):938–0944.

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Beh, E. H., Zheng, F., Dandy, G. C., Maier, H. R., and Kapelan, Z. (2017). Robust optimization of water infrastructure planning under deep uncertainty using metamodels. *Environmental Modelling & Software*, 93:92–105.

Behzadian, K., Kapelan, Z., Savic, D., and Ardeshir, A. (2009). Stochastic sampling design using a multi-objective genetic algorithm and adaptive neural networks. *Environmental Modelling & Software*, 24(4):530–541.

Berglund, T. and McCullough, M. (2011). *Building and Testing with Gradle*. " O'Reilly Media, Inc.".

Bernholdt, D. E., Elwasif, W. R., Kohl, J. A., and Epperly, T. G. (2003). A component architecture for high-performance computing. Technical report, Lawrence Livermore National Lab., CA (US).

Beven, K. (1993). Prophecy, reality and uncertainty in distributed hydrological modelling. *Advances in water resources*, 16(1):41–51.

Beven, K. J. (2011). *Rainfall-runoff modelling: the primer*. John Wiley & Sons.

Bieker, H. P., Slupphaug, O., Johansen, T. A., et al. (2007). Real-time production optimization of oil and gas production systems: A technology survey. *SPE Production & Operations*, 22(04):382–391.

Birrer, A. and Eggenschwiler, T. (1993). Frameworks in the financial engineering domain an experience report. In *European Conference on Object-Oriented Programming*, pages 21–35. Springer.

Blanning, R. W. (1975). The construction and implementation of metamodels. *simulation*, 24(6):177–184.

Blind, M. and Gregersen, J. (2005). Towards an open modelling interface (openmi) the harmonit project. *Advances in Geosciences*, 4:69–74.

Bonnlander, B. V. and Weigend, A. S. (1994). Selecting input variables using mutual information and nonparametric density estimation. In *Proceedings of the 1994 International Symposium on Artificial Neural Networks (ISANN'94)*, pages 42–50. Citeseer.

Bottazzi, M. and Rigon, R. (2018a). Coupling the schymanski-or formula with a multi-layer canopy model. In *EGU General Assembly Conference Abstracts*, volume 20, page 16132.

Bottazzi, M. and Rigon, R. (2018b). Coupling the schymanski-or formula with a multi-layer canopy model. In *International Environmental Modelling and Software Society (iEMSs), Fort Collins, Colorado, USA*.

Boughton, A. (2011). Software metrics.

Bozzini, M. and Rossini, M. (2013). The detection and recovery of discontinuity curves from scattered data. *Journal of Computational and Applied Mathematics*, 240:148–162.

Braun, H. and Weisbrod, J. (1993). Evolving neural feedforward networks. In *Artificial Neural Nets and Genetic Algorithms*, pages 25–32. Springer.

Briand, L. C., Wüst, J., Daly, J. W., and Porter, D. V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software*, 51(3):245–273.

Briand, L. C., Wüst, J., Ikonomovski, S. V., and Lounis, H. (1999). Investigating quality factors in object-oriented designs: an industrial case study. In *Proceedings of the 21st international conference on Software engineering*, pages 345–354. ACM.

Brovelli, M. A. (2006). History of gis. *Laboratrio Geomatica, Politecnico di Milano*.

Burke, B. (2013). *RESTful Java with JAX-RS 2.0: Designing and Developing Distributed Web Services*. " O'Reilly Media, Inc.".

Burkhalter, J. A., Ehlschlaeger, C. R., Morrison, D. M., Myers, N. R., Lu, L., Petit, A., Ouyang, Y., David, O., Serafino, F., Patterson, D., et al. (2018). Integrated analytic simulation tools to support emergency management. In *Next-Generation Analyst VI*, volume 10653, page 106530F. International Society for Optics and Photonics.

Calaway, R., Weston, S., Tenenbaum, D., and Analytics, R. (2015). doparallel: Foreach parallel adaptor for the 'parallel'package. *R package version*, 1(10).

Casulli, V. (2017). A coupled surface-subsurface model for hydrostatic flows under saturated and variably saturated conditions. *International Journal for Numerical Methods in Fluids*, 85(8):449–464.

Chodorow, K. (2013). *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. " O'Reilly Media, Inc.".

Collins, N., Theurich, G., Deluca, C., Suarez, M., Trayanov, A., Balaji, V., Li, P., Yang, W., Hill, C., and Da Silva, A. (2005). Design and implementation of components in the earth system modeling framework. *The International Journal of High Performance Computing Applications*, 19(3):341–350.

Consortium, O. G. et al. (2007). Inc. opengis web processing service (wps) specification, 2007, www document.

Cronshey, R., Theurer, F., and Glenn, R. (1993). Gis-water quality computer model interface: A prototype. In *Second International Conference/Workshop on Integrating Geographic Information Systems and Environmental Modeling, Breckenridge, Colorado*.

Cui, Z., Koren, V., Cajina, N., Voellmy, A., and Moreda, F. (2011). Hydroinformatics advances for operational river forecasting: using graphs for drainage network descriptions. *Journal of Hydroinformatics*, 13(2):181–197.

Dahlgren, T., Epperly, T., Kumfert, G., and Leek, J. (2004). Babel user's guide. *CASC, Lawrence Livermore National Laboratory. version 0.9. 0 edn.(January 2004)*.

Dalla Torre, D. (2019). Swmm through giuh and oms3: the design of stormwater drainage systems leveraging net3. Master's thesis, University of Trento.

Dalla Torre, D., Serafin, F., David, O., and Rigon, R. (2018). Swmm through giuh and oms3: the design of stormwater drainage systems leveraging net3. In *International Environmental Modelling and Software Society (iEMSs), Fort Collins, Colorado, USA*.

Dall'Amico, M., Endrizzi, S., and Tasin, S. (2018). Mysnowmaps: Operative high-resolution real-time snow mapping. In *Proceedings, International Snow Science Workshop, Innsbruck, Austria*, pages 328–332.

Danial, A. (2018). Cloc: Counting lines of code.

Darnell, J. E. and Doolittle, W. (1986). Speculations on the early course of evolution. *Proceedings of the National Academy of Sciences*, 83(5):1271–1275.

Dasgupta, D. and McGregor, D. R. (1992). Designing application-specific neural networks using the structured genetic algorithm. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, pages 87–96. IEEE.

David, O., Ascough Ii, J., Lloyd, W., Green, T., Rojas, K., Leavesley, G., and Ahuja, L. (2013). A software engineering perspective on environmental modeling framework design: The object modeling system. *Environmental Modelling & Software*, 39:201–213.

David, O., Lloyd, W., Ascough, I., James, C., Green, T. R., Olson, K., Leavesley, G., and Carlson, J. R. (2012). Domain specific languages for modeling and simulation: use case oms3. In *International Environmental Modelling and Software Society (iEMSs), Leipzig, Germany*.

David, O., Lloyd, W., Rojas, K., Arabi, M., Geter, F., Ascough, I., James, C., Green, T., Leavesley, G., and Carlson, J. (2014a). Model-as-a-service (maas) using the cloud services innovation platform (csip). In *International Environmental Modelling and Software Society (iEMSs), San Diego, California, USA*.

David, O., Yaegea, L., Rojasd, K., Greenc, T., Kipkaa, H., Lloydab, W., Carlsona, J., Geterd, F., and Ascoughc, J. (2014b). The land management and operations database (lmod). In *International Environmental Modelling and Software Society (iEMSs), San Diego, California, USA*.

Dearle, F. (2010). *Groovy for Domain-Specific Languages*. Packt Publishing Ltd.

Demir, I. and Szczepanek, R. (2017). Optimization of river network representation data models for web-based systems. *Earth and Space Science*, 4(6):336–347.

DePinto, J. V. and Rodgers, P. (1994). Development of geo-wams: A modeling support system for integrating gis with watershed analysis models. *Lake and Reservoir Management*, 9(2):68.

Devia, G. K., Ganasri, B., and Dwarakish, G. (2015). A review on hydrological models. *Aquatic Procedia*, 4:1001–1007.

Dietrich, W. E., Wilson, C. J., Montgomery, D. R., and McKean, J. (1993). Analysis of erosion thresholds, channel networks, and landscape morphology using a digital terrain model. *The Journal of Geology*, 101(2):259–278.

Donatelli, M., Cerrani, I., Fanchini, D., Fumagalli, D., and Rizzoli, A.-E. (2012). Enhancing model reuse via component-centered modeling frameworks: the vision and example realizations. In *Proc. International Congress on Environ. Modell. & Soft., Sixth Biennial Meeting. Leipzig, Germany*.

Dumbser, M., Balsara, D. S., Tavelli, M., and Fambri, F. (2019). A divergence-free semi-implicit finite volume scheme for ideal, viscous, and resistive magnetohydrodynamics. *International Journal for Numerical Methods in Fluids*, 89(1-2):16–42.

Ehlschlaeger, C. R. (1989). Using the aˆ t search algorithm to develop hydrologic models from digital elevation data. In *Proceedings of the International Geographic Information System (IGIS) Symposium, Baltimore, MD*, pages 275–281.

Ehlschlaeger, C. R., Burkhalter, J. A., David, O., Westervelt, J. D., Morrison, D. A., Ouyang, Y., Ross, J., Arabi, M., Patterson, D., Myers, N. R., Carey, B., Bastian, E., Gao, Y., , Lu, L., Serafin, F., Traff, K., Petit, A. M., et al. (2018a). Observations on the implementation of a general purpose spatiotemporal risk analysis system supporting black swan theory. In *University Consortium for Geographic Information Science*.

Ehlschlaeger, C. R., David, O., Ouyang, Y., Westervelt, J. D., Morrison, D. A., Patterson, D., Serafin, F., Lu, L., Burkhalter, J. A., Myers, N. R., et al. (2018b). A computational framework for interoperating uncertainty quantified social system models. In *International Environmental Modelling and Software Society (iEMSs), Fort Collins, Colorado, USA*.

Ehlschlaeger, C. R., David, O., Ouyang, Y., Westervelt, J. D., Morrison, D. A., Patterson, D., Serafin, F., Lu, L., Burkhalter, J. A., Myers, N. R., Petit, A. M., et al. (2018c). Observations on the implementation of a general purpose spatiotemporal risk analysis system supporting black swan theory. In *Conceptualizing a Geospatial Software Institute (GSI)*.

Ehlschlaeger, C. R., DIA, L. M. T. F., Brown, M., Baxter, C. L., Burkhalter, M. J., Calfas, G. W., Copeland, L., CI, M. C., Drigo, M. M., Morrison, A., et al. (2014a). *Socio-Cultural Analysis with the Reconnaissance, Surveillance, and Intelligence Paradigm*. US Army Corps of Engineers, Engineer Research and Development Center ....

Ehlschlaeger, C. R., DIA, L. M. T. F., NGA, M. D. E. B., NGA, E. B., and Brandenberger, M. J. (2014b). *Understanding Megacities with the Reconnaissance, Surveillance, and Intelligence Paradigm*. US Army Corps of Engineers, Engineer Research and Development Center ....

Fatichi, S., Vivoni, E. R., Ogden, F. L., Ivanov, V. Y., Mirus, B., Gochis, D., Downer, C. W., Camporese, M., Davison, J. H., Ebel, B., et al. (2016). An overview of current applications, challenges, and future trends in distributed process-based models in hydrology. *Journal of Hydrology*, 537:45–60.

Fielding, R. T. and Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, USA.

Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150.

Floreano, D., Dürr, P., and Mattiussi, C. (2008). Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62.

Formetta, G. (2013). *Hydrological modelling with components: the OMS3 NewAge-JGrass system*. PhD thesis, University of Trento.

Formetta, G., Antonello, A., Franceschi, S., David, O., and Rigon, R. (2014a). Hydrological modelling with components: A gis-based open-source framework. *Environmental Modelling & Software*, 55:190–200.

Formetta, G., Bancheri, M., David, O., and Rigon, R. (2016a). Performance of site-specific parameterizations of longwave radiation. *Hydrology and Earth System Sciences*, 20(11):4641–4654.

Formetta, G., Capparelli, G., David, O., Green, T. R., and Rigon, R. (2016b). Integration of a three-dimensional process-based hydrological model into the object modeling system. *Water*, 8(1):12.

Formetta, G., Capparelli, G., and Versace, P. (2016c). Evaluating performance of simplified physically based models for shallow landslide susceptibility. *Hydrology and Earth System Sciences*, 20(11):4585–4603.

Formetta, G., Kampf, S., David, O., and Rigon, R. (2013a). The cache la poudre river basin snow water equivalent modeling with newage-jgrass. *Geoscientific Model Development Discussions*, 6(3).

Formetta, G., Kampf, S. K., David, O., and Rigon, R. (2014b). Snow water equivalent modeling components in newage-jgrass. *Geoscientific Model Development*, 7(3):725–736.

Formetta, G., Mantilla, R., Franceschi, S., Antonello, A., and Rigon, R. (2011). The jgrass-newage system for forecasting and managing the hydrological budgets at the basin scale: models of flow generation and propagation/routing. *Geoscientific Model Development*, 4(4):943–955.

Formetta, G., Rigon, R., Chávez, J., and David, O. (2013b). Modeling shortwave solar radiation using the jgrass-newage system. *Geoscientific Model Development*, 6(4):915–928.

Foster, G. (2005). Draft science documentation, revised universal soil loss equation version 2 (rusle2). washington (dc): Agricultural research service. *US Department of Agriculture*.

Foster, G., Yoder, D., McCool, D., Weesies, G., Toy, T., Wagner, L., et al. (2000). Improvements in science in rusle2. *Improvements in science in RUSLE2.*, pages 1–19.

Foster, G., Yoder, D., Weesies, G., and Toy, T. (2001). The design philosophy behind rusle2: Evolution of an empirical model. In *Soil Erosion*, page 95. American Society of Agricultural and Biological Engineers.

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. *http://www. martinfowler. com/articles/injection. html.*

Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

Freeman, E., Robson, E., Bates, B., and Sierra, K. (2004). *Head first design patterns*. " O'Reilly Media, Inc.".

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics New York, NY, USA:.

Fullmer, B. and Miikkulainen, R. (1992). Using marker-based genetic encoding of neural networks to evolve finite-state behaviour. In *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 255–262. MIT Press.

Gachet, A. (2003). A new component in the classification of dss development tools. *Journal of Decision Systems*, 12(3-4):271–281.

Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

Gironás, J., Roesner, L. A., Rossman, L. A., and Davis, J. (2010). A new applications manual for the storm water management model (swmm). *Environmental Modelling & Software*, 25(6):813–814.

Goetz, B. and Peierls, T. (2006). *Java concurrency in practice*. Pearson Education.

Goldberg, D. E., Richardson, J., et al. (1987). Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Hillsdale, NJ: Lawrence Erlbaum.

Goran, W. D., Dvorak, W. E., Warren, L. V., and Webster, R. D. (1983). Fort hood geographic information system: Pilot system development and user instructions. Technical report, CONSTRUCTION ENGINEERING RESEARCH LAB (ARMY) CHAMPAIGN IL.

Gordon, R. (1998). *Essential JNI: Java Native Interface*. Prentice-Hall, Inc.

Govindaraju, R. S. and Rao, A. R. (2000a). *Artificial Neural Networks in Hydrology*, volume 36. Kluwer Academic Publishers.

Govindaraju, R. S. and Rao, A. R. (2000b). *Artificial Neural Networks in Hydrology*, volume 36. Kluwer Academic Publishers.

Grasso, S., Giuzio, L., Viggiano, D., and Manfreda, S. (2017). Sviluppo di un sistema web-gis per l'early warning di criticità pluviometrica. In *Proceedings of the 21 Conferenza Nazionale ed EXPO.*, pages 641–650.

Green, T. R., Erskine, R. H., Ascough II, J. C., Vandenberg, B., Pfennig, B., Kipka, H., David, O., and Coleman, M. L. (2014). Agroecosystem-watershed (ages-w) model delineation and scaling. In *Proceedings of the Seventh International Congress on Environmental Modelling and Software. June*, pages 15–19.

Green, T. R., Erskine, R. H., Coleman, M. L., David, O., Ascough, J. C., and Kipka, H. (2015). The agroecosystem (ages) response-function model simulates layered soil-water dynamics in semiarid colorado: Sensitivity and calibration. *Vadose Zone Journal*, 14(8).

Green, T. R., Kipka, H., David, O., and McMaster, G. S. (2018a). Where is the usa corn belt, and how is it changing? *Science of the Total Environment*, 618:1613–1618.

Green, T. R., Kipka, H., Tomer, M., McMaster, G. S., Beeson, P., Lighthart, N., David, O., Arabi, M., and Ascough, J. (2018b). Streamflow rates and nitrogen loads in the south fork iowa river, usa: Simulations with spatial interactions and tile drainage indicate high manure applications. *Agricultural Water Management*.

Gregersen, J., Gijsbers, P., and Westen, S. (2007). Openmi: Open modelling interface. *Journal of hydroinformatics*, 9(3):175–191.

Gruau, F. (1993). Genetic synthesis of modular neural networks. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 318–325. Morgan Kaufmann Publishers Inc.

Gruau, F., Whitley, D., and Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the 1st annual conference on genetic programming*, pages 81–89. MIT Press.

Grübsch, M. and David, O. (2001). How to divide a catchment to conquer its parallel processing. an efficient algorithm for the partitioning of water catchments. *Mathematical and computer modelling*, 33(6-7):723–731.

Guo, Z., Zhao, W., Lu, H., and Wang, J. (2012). Multi-step forecasting for wind speed using a modified emd-based artificial neural network model. *Renewable Energy*, 37(1):241–249.

Harbaugh, A. W., Banta, E. R., Hill, M. C., and McDonald, M. G. (2000). Modflow-2000, the u. s. geological survey modular ground-water model-user guide to modularization concepts and the ground-water flow process. *Open-file Report. U. S. Geological Survey*, page 134.

Hay, L., Knapp, L., and Bromberg, J. (1993). Integrating geographic information systems, scientific visualization systems, statistics, and an orographic precipitation model for a hydro-climatic study of the gunnison river basin, southwestern colorado. In *Proceedings of the Second International Conference/Workshop on Integrating Geographic Information Systems and Environmental Modeling, Breckenridge, CO*.

Haykin, S. and Lippmann, R. (1994). Neural networks, a comprehensive foundation. *International journal of neural systems*, 5(4):363–364.

Heaton, J. (2015). Encog: library of interchangeable machine learning models for java and c#. *Journal of Machine Learning Research*, 16:1243–1247.

Heckmann, T., Schwanghart, W., and Phillips, J. D. (2015). Graph theory—recent developments of its application in geomorphology. *Geomorphology*, 243:130–146.

Hijmans, R. J., van Etten, J., Cheng, J., Mattiuzzi, M., Sumner, M., Greenberg, J. A., Lamigueiro, O. P., Bevan, A., Racine, E. B., Shortridge, A., et al. (2015). Package 'raster'. *R package*.

Hill, C., DeLuca, C., Suarez, M., Da Silva, A., et al. (2004). The architecture of the earth system modeling framework. *Computing in Science & Engineering*, 6(1):18.

Hluchy, L., Froehlich, D., Tran, V. D., Astalos, J., Dobrucky, M., and Nguyen, G. T. (2001). Parallel numerical solution for flood modeling systems. In *International Conference on Parallel Processing and Applied Mathematics*, pages 485–492. Springer.

Hooshyar, M., Wang, D., Kim, S., Medeiros, S. C., and Hagen, S. C. (2016). Valley and channel networks extraction based on local topographic curvature and k-means clustering of contours. *Water Resources Research*, 52(10):8081–8102.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558.

Howard, A. D. (1994). A detachment-limited model of drainage basin evolution. *Water resources research*, 30(7):2261–2285.

Hrachowitz, M. and Clark, M. P. (2017). Hess opinions: The complementary merits of competing modelling philosophies in hydrology. *Hydrology and Earth System Sciences*, 21(8):3953–3973.

Ijjasz-Vasquez, E. J. and Bras, R. L. (1995). Scaling regimes of local slope versus contributing area in digital elevation models. *Geomorphology*, 12(4):299–311.

Iuhasz, G., Munteanu, V. I., and Negru, V. (2013). Data mining considerations for knowledge acquisition in real time strategy games. In *2013 IEEE 11th International Symposium on Intelligent Systems and Informatics (SISY)*, pages 331–336. IEEE.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An introduction to statistical learning*, volume 112. Springer.

Jasiewicz, J. and Metz, M. (2011). A new grass gis toolkit for hortonian analysis of drainage networks. *Computers & Geosciences*, 37(8):1162–1173.

Jha, S., Katz, D. S., Luckow, A., Merzky, A., and Stamou, K. (2011). Understanding scientific applications for cloud environments. *Cloud computing: principles and paradigms*, pages 345–371.

Jin, Y., Olhofer, M., and Sendhoff, B. (2002). A framework for evolutionary optimization with approximate fitness functions. *IEEE Transactions on evolutionary computation*, 6(5):481–494.

Johnson, R. E. (1992). Documenting frameworks using patterns. In *OOPSLA*, volume 92, pages 63–76. Citeseer.

Johnson, R. E., McConnell, C., and Lake, J. M. (1992). The rtl system: A framework for code optimization. In *Code Generation—Concepts, Tools, Techniques*, pages 255–274. Springer.

Jones, E., Oliphant, T., and Peterson, P. (2014). {SciPy}: Open source scientific tools for {Python}.

Julien, P. Y. and Saghafian, B. (1991). Casc2d user's manual: a two-dimensional watershed rainfall-runoff model. *CER; 90/91-12*.

Kalyanapu, A. J., Shankar, S., Pardyjak, E. R., Judi, D. R., and Burian, S. J. (2011). Assessment of gpu computational enhancement to a 2d flood model. *Environmental Modelling & Software*, 26(8):1009–1016.

Khu, S.-T. and Werner, M. G. (2003). Reduction of monte-carlo simulation runs for uncertainty estimation in hydrological modelling. *Hydrology and Earth System Sciences Discussions*, 7(5):680–692.

Kira, K. and Rendell, L. A. (1992). A practical approach to feature selection. In *Machine Learning Proceedings 1992*, pages 249–256. Elsevier.

Kleijnen, J. P. (1975). A comment on blanning's "metamodel for sensitivity analysis: the regression metamodel in simulation". *Interfaces*, 5(3):21–23.

Kourakos, G. and Mantoglou, A. (2009). Pumping optimization of coastal aquifers based on evolutionary algorithms and surrogate modular neural network models. *Advances in water resources*, 32(4):507–521.

Krishnan, R. and Ciesielski, V. B. (1994). Delta-gann: A new approach to training neural networks using genetic algorithms. In *University of Queensland*. Citeseer.

Krummel, J., Dunn, C., Eckert, T., and Ayers, A. (1996). A technology to analyze spatiotemporal landscape dynamics: Application to cadiz township (wisconsin). *1996a) op. cit*, pages 169–174.

Laflen, J. M., Lane, L. J., and Foster, G. R. (1991). Wepp: A new generation of erosion prediction technology. *Journal of Soil and Water Conservation*, 46(1):34–38.

Langley, P. (1994). Selection of relevant features in machine learning. In *Proceedings of the AAAI Fall symposium on relevance*, pages 1–5.

Lashermes, B., Foufoula-Georgiou, E., and Dietrich, W. E. (2007). Channel network extraction from high resolution topography using wavelets. *Geophysical Research Letters*, 34(23).

Leavesley, G., Markstrom, S., Viger, R., and Hay, L. (2005). Usgs modular modeling system (mms)-precipitation-runoff modeling system (prms) mms-prms. *Singh, V., and Frevert, D., CRC Press, Boca Raton, USA*, pages 159–177.

Lee, C.-H. and Kim, J.-H. (1996). Evolutionary ordered neural network with a linked-list encoding scheme. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 665–669. IEEE.

Leonard, R., Knisel, W., and Still, D. (1987). Gleams: Groundwater loading effects of agricultural management systems. *Transactions of the ASAE*, 30(5):1403–1418.

Li, T., Wang, G., Chen, J., and Wang, H. (2011). Dynamic parallelization of hydrological model simulations. *Environmental Modelling & Software*, 26(12):1736–1746.

Liaw, A., Wiener, M., et al. (2002). Classification and regression by randomforest. *R news*, 2(3):18–22.

Lindsay, J. B. (2005). The terrain analysis system: A tool for hydro-geomorphic applications. *Hydrological Processes: An International Journal*, 19(5):1123–1130.

Liong, S.-Y., Khu, S.-T., and Chan, W.-T. (2001). Derivation of pareto front with genetic algorithm and neural network. *Journal of Hydrologic Engineering*, 6(1):52–61.

Liu, J., Zhu, A.-X., Qin, C.-Z., Wu, H., and Jiang, J. (2016). A two-level parallelization method for distributed hydrological models. *Environmental modelling & software*, 80:175–184.

Lloyd, W., David, O., Ascough, I., James, C., Green, T., Carlson, J., Lyon, J., and Rojas, K. (2012). The cloud services innovation platform-enabling service-based environmental modelling using infrastructure-as-a-service cloud computing. In *International Environmental Modelling and Software Society (iEMSs), Leipzig, Germany*.

Lloyd, W., David, O., Ascough, J., Rojas, K. W., Carlson, J. R., Leavesley, G., Krause, P., Green, T. R., and Ahuja, L. (2011). Environmental modeling framework invasiveness: Analysis and implications. *Environmental Modelling & Software*, 26(10):1240–1250.

Lu, L., Wang, X., Ouyang, Y., Roningen, J., Myers, N., and Calfas, G. (2018). Vulnerability of interdependent urban infrastructure networks: Equilibrium after failure propagation and cascading impacts. *Computer-Aided Civil and Infrastructure Engineering*, 33(4):300–315.

Maniezzo, V. (1994). Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on neural networks*, 5(1):39–53.

Martin, R. C. (2007). Professionalism and test-driven development. *Ieee Software*, 24(3):32–36.

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

McCreary, C. and Reed, A. (1993). A graph parsing algorithm and implementation. *Tech. Rpt. TR-93-04, Dept. of Comp. Sci and Eng., Auburn U.*

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

Meyer, M. (2014). Continuous integration and its tools. *IEEE software*, 31(3):14–16.

Miller, J. C. and Maloney, C. J. (1963). Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63.

MongoDB (2019a). Mongodb: Aggregation. docs.mongodb.com/manual/aggregation/#single-purpose-aggregation-operations. Accessed: 2019-01-13.

MongoDB (2019b). Mongodb: Map-reduce. docs.mongodb.com/manual/core/map-reduce/. Accessed: 2019-01-13.

Montgomery, D. R. (1999). Process domains and the river continuum 1. *JAWRA Journal of the American Water Resources Association*, 35(2):397–410.

Montgomery, D. R. and Dietrich, W. E. (1988). Where do channels begin? *Nature*, 336(6196):232.

Montgomery, D. R. and Dietrich, W. E. (1989). Source areas, drainage density, and channel initiation. *Water Resources Research*, 25(8):1907–1918.

Montgomery, D. R. and Foufoula-Georgiou, E. (1993). Channel network source representation using digital elevation models. *Water Resources Research*, 29(12):3925–3934.

Moore, A., Holzworth, D., Herrmann, N., Huth, N., and Robertson, M. (2007). The common modelling protocol: A hierarchical framework for simulation of agricultural and environmental systems. *Agricultural Systems*, 95(1-3):37–48.

Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.".

Narendra, P. M. and Fukunaga, K. (1977). A branch and bound algorithm for feature subset selection. *IEEE Transactions on computers*, pages 917–922.

Newman, S. (2015). *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

Oliphant, T. E. (2006). *A guide to NumPy*, volume 1. Trelgol Publishing USA.

Opitz, D. W. and Shavlik, J. W. (1997). Connectionist theory refinement: Genetically searching the space of network topologies. *Journal of Artificial Intelligence Research*, 6:177–209.

Ovatman, T., Weigert, T., and Buzluca, F. (2011). Exploring implicit parallelism in class diagrams. *Journal of Systems and Software*, 84(5):821–834.

O'Hagan, A. (2006). Bayesian analysis of computer code outputs: A tutorial. *Reliability Engineering & System Safety*, 91(10-11):1290–1300.

Paniconi, C. and Putti, M. (2015). Physically based modeling in catchment hydrology at 50: Survey and outlook. *Water Resources Research*, 51(9):7090–7129.

Passalacqua, P., Do Trung, T., Foufoula-Georgiou, E., Sapiro, G., and Dietrich, W. E. (2010a). A geometric framework for channel network extraction from lidar: Nonlinear diffusion and geodesic paths. *Journal of Geophysical Research: Earth Surface*, 115(F1).

Passalacqua, P., Tarolli, P., and Foufoula-Georgiou, E. (2010b). Testing space-scale methodologies for automatic geomorphic feature extraction from lidar in a complex mountainous landscape. *Water resources research*, 46(11).

Pebesma, E. J. and Wesseling, C. G. (1998). Gstat: a program for geostatistical modelling, prediction and simulation. *Computers & Geosciences*, 24(1):17–31.

Pechlivanidis, I., Jackson, B., McIntyre, N., and Wheater, H. (2011). Catchment scale hydrological modelling: a review of model types, calibration approaches and uncertainty analysis methods in the context of recent developments in technology and applications. *Global NEST journal*, 13(3):193–214.

Peckham, S. D. (1995). *Self-similarity in the three-dimensional geometry and dynamics of large river basins*. PhD thesis, University of Colorado.

Peckham, S. D., Hutton, E. W., and Norris, B. (2013). A component-based approach to integrated modeling in the geosciences: The design of csdms. *Computers & Geosciences*, 53:3–12.

Peckham, S. D., Stoica, M., Jafarov, E., Endalamaw, A., and Bolton, W. R. (2017). Reproducible, component-based modeling with topoflow, a spatial hydrologic modeling toolkit. *Earth and Space Science*, 4(6):377–394.

Pelletier, J. D. (2013). A robust, two-parameter method for the extraction of drainage networks from high-resolution digital elevation models (dems): Evaluation using synthetic and real-world dems. *Water Resources Research*, 49(1):75–89.

Phillips, J. D., Schwanghart, W., and Heckmann, T. (2015). Graph theory in the geosciences. *Earth-Science Reviews*, 143:147–160.

Plugge, E., Hows, D., Membrey, P., and Hawkins, T. (2015). *The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB*. Apress.

Porporato, A. and Ridolfi, L. (2001). Multivariate nonlinear prediction of river flows. *Journal of Hydrology*, 248(1-4):109–122.

Pudil, P., Novovičová, J., and Kittler, J. (1994). Floating search methods in feature selection. *Pattern recognition letters*, 15(11):1119–1125.

Pujol, J. C. F. and Poli, R. (1998). Evolving the topology and the weights of neural networks using a dual representation. *Applied Intelligence*, 8(1):73–84.

Qu, Y. and Duffy, C. J. (2007). A semidiscrete finite volume formulation for multiprocess watershed simulation. *Water Resources Research*, 43(8).

Quesnel, G., Duboz, R., and Ramat, É. (2009). The virtual laboratory environment–an operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simulation Modelling Practice and Theory*, 17(4):641–653.

Ramos-Pollán, R., Guevara-López, M. Á., and Oliveira, E. (2012). A software framework for building biomedical machine learning classifiers through grid computing resources. *Journal of medical systems*, 36(4):2245–2257.

Rao, P. (2005). A parallel rma2 model for simulating large-scale free surface flows. *Environmental Modelling & Software*, 20(1):47–53.

Razavi, S., Tolson, B. A., and Burn, D. H. (2012a). Numerical assessment of metamodelling strategies in computationally intensive optimization. *Environmental Modelling & Software*, 34:67–86.

Razavi, S., Tolson, B. A., and Burn, D. H. (2012b). Review of surrogate modeling in water resources. *Water Resources Research*, 48(7).

Regis, R. G. and Shoemaker, C. A. (2005). Constrained global optimization of expensive black box functions using radial basis functions. *Journal of Global optimization*, 31(1):153–171.

Renard, K. G., Foster, G. R., Weesies, G., McCool, D., Yoder, D., et al. (1997). *Predicting soil erosion by water: a guide to conservation planning with the Revised Universal Soil Loss Equation (RUSLE)*, volume 703. United States Department of Agriculture Washington, DC.

Rewerts, C. C. and Engel, B. (1991). Answers on grass: Integrating a watershed simulation with a gis. *Paper-American Society of Agricultural Engineers (USA). no. 91-2621.*

Richardson, C. (2006). *POJOs in Action: Developing Enterprise Applications with Lightweight Frameworks*. Manning Publications Co.

Richardson, L. and Ruby, S. (2008). *RESTful web services*. " O'Reilly Media, Inc.".

Rigon, R. (2014). Jgrass-newage history - version zero and version one. Accessed: 2019-01-13.

Rigon, R., Bancheri, M., Formetta, G., and de Lavenne, A. (2016). The geomorphological unit hydrograph from a historical-critical perspective. *Earth Surface Processes and Landforms*, 41(1):27–37.

Rigon, R., Bertoldi, G., and Over, T. M. (2006a). Geotop: A distributed hydrological model with coupled water and energy budgets. *Journal of Hydrometeorology*, 7(3):371–388.

Rigon, R., Ghesla, E., Tiso, C., and Cozzini, A. (2006b). The horton machine: a system for dem analysis the reference manual. *Università degli Studi di Trento*.

Rigon, R., Rodriguez-Iturbe, I., and Rinaldo, A. (1998). Feasible optimality implies hack's law. *Water resources research*, 34(11):3181–3189.

Rigon, R., Tubini, N., Bottazzi, M., Serafin, F., and Marialaura, B. (2018). Experiences in using geotop model (pde based) and geoframe-newage system (odes based). In *Integrated Hydrosystem Modelling*.

Rizzoli, A., Leavesley, G., Ascough, I., Argent, R., Athanasiadis, I., Brilhante, V., Claeys, F., David, O., Donatelli, M., Gijsbers, P., et al. (2008). Integrated modelling frameworks for environmental assessment and decision support. *State of the Art and new perspective*.

Rizzoli, A., Svensson, M., Rowe, E., Donatelli, M., Muetzelfeldt, R., van der Wal, T., van Evert, F., and Villa, F. (2006). Modelling framework (seamframe) requirements. Technical report, SEAMLESS.

Robinson, T., Eldred, M., Willcox, K., and Haimes, R. (2008). Surrogate-based optimization using multifidelity models with variable parameterization and corrected space mapping. *Aiaa Journal*, 46(11):2814–2822.

Robinson, T. D. (2007). *Surrogate-based optimization using multifidelity models with variable parameterization*. PhD thesis, Massachusetts Institute of Technology.

Roman, D., Schade, S., Berre, A., Bodsberg, N. R., and Langlois, J. (2009). Model as a service (maas). In *AGILE Workshop: Grid Technologies for Geospatial Applications, Hannover, Germany*.

Rossman, L. A. (2010). *Storm water management model user's manual, version 5.0*. National Risk Management Research Laboratory, Office of Research and ....

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

Sedgewick, R. and Wayne, K. (2011). *Algorithms*. Addison-Wesley Professional.

Serafin, F., Bancheri, M., David, O., and Rigon, R. (2017). On complex networks representation and computation of hydrologycal quantities. In *AGU Fall Meeting Abstracts*.

Serafin, F., Bancheri, M., David, O., and Rigon, R. (2018a). On complex network computation of mountain catchments. In *5th IAHR European Congress - New challenges in hydraulic research and engineering*.

Serafin, F., David, O., Dozier, A. Q., Carlson, J. R., and Ehlschlaeger, C. R. (2018b). Framework-enabled meta-modeling. In *International Environmental Modelling and Software Society (iEMSs), Fort Collins, CO, USA*.

Serafin, F., Westervelt, J. D., Ehlschlaeger, C. R., David, O., Liqun, L., Petit, A. M., Zhoutong, J., and Yanfeng, O. (2018c). R and python annotation bindings for oms. In *International Environmental Modelling and Software Society (iEMSs), Fort Collins, CO, USA*.

Sexton, R. S., Dorsey, R. E., and Johnson, J. D. (1998). Toward global optimization of neural networks: a comparison of the genetic algorithm and backpropagation. *Decision Support Systems*, 22(2):171–185.

Shrestha, D., Kayastha, N., and Solomatine, D. (2009). A novel approach to parameter uncertainty analysis of hydrological models using neural networks. *Hydrology and Earth System Sciences*, 13(7):1235–1248.

Smith, L., Beckman, R., and Baggerly, K. (1995). Transims: Transportation analysis and simulation system. Technical report, Los Alamos National Lab., NM (United States).

Sofia, G., Tarolli, P., Cazorzi, F., and Dalla Fontana, G. (2011). An objective approach for feature extraction: distribution analysis and statistical descriptors for scale choice and channel network identification. *Hydrology and earth system sciences*, 15(5):1387–1402.

Spear, R. and Hornberger, G. (1980). Eutrophication in peel inlet—ii. identification of critical uncertainties via generalized sensitivity analysis. *Water Research*, 14(1):43–49.

Srinivasan, R. (1992). *Spatial Decision Support System for Assessing Agricultural Non-Point Source Pollution Using GIS*. PhD thesis, Purdue University.

Srinivasan, R., Ramanarayanan, T. S., Arnold, J. G., and Bednarz, S. T. (1998). Large area hydrologic modeling and assessment part ii: model application 1. *JAWRA Journal of the American Water Resources Association*, 34(1):91–101.

Srinivasulu, S. and Jain, A. (2009). Rainfall-runoff modelling: Integrating available data and modern techniques. In *Practical Hydroinformatics*, pages 59–70. Springer.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

Stevens, W. P., Myers, G. J., and Constantine, L. L. (1974). Structured design. *IBM Systems Journal*, 13(2):115–139.

Swiler, L. P. and Giunta, A. A. (2007). Aleatory and epistemic uncertainty quantification for engineering applications. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); Sandia ....

Taheri, T. (2010). Benchmarking and comparing encog, neuroph and joone neural networks. `http://goo.gl/A56iyx`. Accessed: 2018-08-25.

Tan, M., Hartley, M., Bister, M., and Deklerck, R. (2009). Automated feature selection in neuroevolution. *Evolutionary Intelligence*, 1(4):271–292.

Tarboton, D. G. (1997). A new method for the determination of flow directions and upslope areas in grid digital elevation models. *Water resources research*, 33(2):309–319.

Tarboton, D. G., Bras, R. L., and Rodriguez-Iturbe, I. (1991). On the extraction of channel networks from digital elevation data. *Hydrological processes*, 5(1):81–100.

Team, R. D., Hanson, J., Ahuja, L., Shaffer, M., Rojas, K., DeCoursey, D., Farahani, H., and Johnson, K. (1998). Rzwqm: Simulating the effects of management on water quality and crop production. *Agricultural Systems*, 57(2):161–195.

Tomer, M., Moorman, T., and Rossi, C. (2008). Assessment of the iowa river's south fork watershed: part 1. water quality. *journal of soil and water conservation*, 63(6):360–370.

Tran, V. D. and Hluchy, L. (2004). Parallelizing flood models with mpi: Approaches and experiences. In *International Conference on Computational Science*, pages 425–428. Springer.

Tubini, N., Rigon, R., Gruber, S., and Casulli, V. (2018). New insights in modeling coupled surface-subsurface flow in glacial and periglacial catchments. In *EGU General Assembly Conference Abstracts*, volume 20, page 13980.

Tubini, N., Serafin, F., Gruber, S., Casulli, V., and Rigon, R. (2017). New insights in permafrost modelling. In *EGU General Assembly Conference Abstracts*, volume 19, page 4870.

Urbanek, S. (2003). Rserve–a fast way to provide r functionality to applications. In *PROC. OF THE 3RD INTERNATIONAL WORKSHOP ON DISTRIBUTED STATISTICAL COMPUTING (DSC 2003), ISSN 1609-395X, EDS.: KURT HORNIK, FRIEDRICH LEISCH & ACHIM ZEILEIS, 2003 (HTTP://ROSUDA.ORG/RSERVE*. Citeseer.

USDA, N. (1987). Estimating runoff and peak discharge.

Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages. *Centrum voor Wiskunde en Informatika*, 5:12.

Van Etten, J. (2012). R package gdistance: distances and routes on geographical grids (version 1.1-4).

Viana, F. A. and Haftka, R. T. (2008). Using multiple surrogates for metamodeling. In *Proceedings of the 7th ASMO-UK/ISSMO International conference on engineering design optimization*, pages 1–18.

Vivoni, E. R., Mniszewski, S., Fasel, P., Springer, E., Ivanov, V., and Bras, R. (2005). Parallelization of a fully-distributed hydrologic model using sub-basin partitioning. *Eos Trans. AGU*, 86(52).

Vlissides, J. M. and Linton, M. A. (1990). Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems (TOIS)*, 8(3):237–268.

Wagener, T., Boyle, D. P., Lees, M. J., Wheater, H. S., Gupta, H. V., and Sorooshian, S. (2001). A framework for development and application of hydrological models. *Hydrology and Earth System Sciences*, 5(1):13–26.

Wagener, T., Lees, M., and Wheater, H. (2000). Reducing conceptual rainfall-runoff modelling uncertainty. In *Proc. of Workshop on "Runoff Generation and Implications for River Basin Modelling", Freiburg, Germany*.

Wagener, T., Wheater, H., and Gupta, H. V. (2004). *Rainfall-runoff modelling in gauged and ungauged catchments*. World Scientific.

Wang, H., Fu, X., Wang, G., Li, T., and Gao, J. (2011). A common parallel computing framework for modeling hydrological processes of river basins. *Parallel Computing*, 37(6):302–315.

Wang, H., Schmitt, J., and Ciucu, F. (2013). Performance modelling and analysis of unreliable links with retransmissions using network calculus. In *Teletraffic Congress (ITC), 2013 25th International*, pages 1–9. IEEE.

Watson, J. D., Hopkins, N. H., Roberts, J. W., Steitz, J. A., and Weiner, A. M. (1988). Molecular biology of the gene. In *Molecular biology of the gene*. Benjamin/Cummings Publishing.

Westervelt, J. (2001). *Simulation modeling for watershed management*. Springer Science & Business Media.

Westervelt, J., BenDor, T., and Sexton, J. (2011). A technique for rapidly forecasting regional urban growth. *Environment and Planning B: Planning and Design*, 38(1):61–81.

Westervelt, J. et al. (1991). Introduction to grass 4. *GRASS Information Center, US Army CERL, Champaign, Illinois, US July*.

Westervelt, J. D., Ehlschlaeger, C. R., Burkhalter, J. A., and Baxter, C. L. (2017). Sparse-data forecasting of megacity growth. *Military Operations Research*, 22(3):21–34.

Wheater, H., Ballard, C., Bulygina, N., McIntyre, N., and Jackson, B. (2012). Modelling environmental change: quantification of impacts of land use and land management change on uk flood risk. In *System Identification, Environmental Modelling, and Control System Design*, pages 449–481. Springer.

Whiteson, S., Stone, P., Stanley, K. O., Miikkulainen, R., and Kohl, N. (2005). Automatic feature selection in neuroevolution. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1225–1232. ACM.

Wikipedia, c. (2004). Software metrics. https://en.wikipedia.org/wiki/Software_metric. Accessed: 2018-07-12.

Wikipedia, c. (2018). Sensitivity analysis. https://en.wikipedia.org/wiki/Sensitivity_analysis. Accessed: 2018-08-05.

Willcox, K. and Peraire, J. (2002). Balanced model reduction via the proper orthogonal decomposition. *AIAA journal*, 40(11):2323–2330.

Williams, J., Arnold, J., Kiniry, J., Gassman, P., and Green, C. (2008). History of model development at temple, texas. *Hydrological sciences journal*, 53(5):948–960.

Williams, J., Nicks, A., and Arnold, J. (1985). Simulator for water resources in rural basins. *Journal of Hydraulic Engineering*, 111(6):970–986.

Wood, J. (2009). The landserf manual. *User Guide for LandSerf*, 23.

Xie, S., An, K., and Ouyang, Y. (2019). Planning facility location under generally correlated facility disruptions: Use of supporting stations and quasi-probabilities. *Transportation Research Part B: Methodological*, 122:115–139.

Yan, S. and Minsker, B. (2006). Optimal groundwater remediation design using an adaptive neural network genetic algorithm. *Water Resources Research*, 42(5).

Yan, S. and Minsker, B. (2010). Applying dynamic surrogate models in noisy genetic algorithms to optimize groundwater remediation designs. *Journal of Water Resources Planning and Management*, 137(3):284–292.

Yao, X. and Liu, Y. (1998). Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83–90.

Yapo, P. O., Gupta, H. V., and Sorooshian, S. (1998). Multi-objective global optimization for hydrologic models. *Journal of hydrology*, 204(1-4):83–97.

Young, P. C. and Leedal, D. (2013). Data-based mechanistic modelling and the emulation of large environmental system models. *Environmental Modelling: Finding Simplicity in Complexity*, pages 111–131.

Young, R., Onstad, C., Bosch, D., and Anderson, W. (1989). Agnps: A nonpoint-source pollution model for evaluating agricultural watersheds. *Journal of soil and water conservation*, 44(2):168–173.

Yourdon, E. and Constantine, L. L. (1979). *Structured design: Fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc.

Yu, L., Wang, S., and Lai, K. K. (2008). Forecasting crude oil price with an emd-based neural network ensemble learning paradigm. *Energy Economics*, 30(5):2623–2635.

Zhang, B. and Govindaraju, R. S. (2000). Prediction of watershed runoff using bayesian concepts and modular neural networks. *Water Resources Research*, 36(3):753–762.

Zhang, B.-T. and Muhlenbein, H. (1993). Evolving optimal neural networks using genetic algorithms with occam's razor. *Complex systems*, 7(3):199–220.

Zhang, X., Srinivasan, R., and Van Liew, M. (2009). Approximating swat model using artificial neural network and support vector machine 1. *JAWRA Journal of the American Water Resources Association*, 45(2):460–474.

Zou, G., Zhang, B., Zheng, J., Li, Y., and Ma, J. (2012). Maas: Model as a service in cloud computing and cyber-i space. In *2012 IEEE 12th International Conference on Computer and Information Technology*, pages 1125–1130. IEEE.

Zou, R., Lung, W.-S., and Wu, J. (2009). Multiple-pattern parameter identification and uncertainty analysis approach for water quality modeling. *Ecological Modelling*, 220(5):621–629.

Conceptual and physically based environmental simulation models as products of research environments efforts became complex software over time in order to allow describing the behaviour of natural phenomena more accurately.
Results from these models are considered accurate but often require to operate an entire system of modeling resources with dedicated knowledge, an extensive set up, and sometimes significant computational time. Model complexity limits wide model adaptation among consultants because of lower available technical resources and capabilities. However, models should be ubiquitous to use in both research and consulting environments.
This dissertation aims to address and alleviate two aspects of research model complexity: 1) for researchers, the model design complexity with respect to its internal software structure and 2) for consultants, the model application complexity with respect to data and parameter setup, runtime requirements, and proper model infrastructure setup. The first contribution provides modeling design and implementation support by managing interacting modeling solutions as “Directed Acyclic Graph”, while the second one helps to create surrogate models of complex physical models as a streamlined process.
Both contributions are implemented within the Object Modeling System/Cloud Service Integration Platform modeling framework and infrastructure and were applied in various studies.

Francesco Serafin receives a Master's Degree in Environmental Engineering from University of Trento, Italy in 2014. His main research topic focuses on environmental modeling frameworks development to 1) simplify dissemination and use of complex environmental models, and 2) accommodate innovative modeling practices and facilitate model implementation, integration and maintenance. During his PhD studies, he strengthened his environmental engineering background and deepened his knowledge in computer science related topics. He investigated graph theory and complex networks and their flexible adaptation to environmental modeling applications. He studied data-driven surrogate modeling systems and their automated generation at a framework level. He also spent one and a half year at Colorado State University to learn architectural concepts and design of the state-of-art frameworks Object Modeling System v3 and Cloud Service Integration Platform.